Lightweight urban computation interchange (LUCI): a system to couple heterogeneous simulations and views

Spat. Inf. Res.
DOI 10.1007/s41324-016-0025-y

Lightweight urban computation interchange (LUCI): a system to couple heterogeneous simulations and views

Lukas Treyer[1] · Bernhard Klein[2] · Reinhard König[1] · Christine Meixner[1]

Received: 10 February 2016 / Revised: 5 May 2016 / Accepted: 10 May 2016

Abstract In this paper we introduce LUCI, a lightweight urban computation interchange system, designed to bring the advantages of calculation and content co-ordination system to small planning and design groups by the means of an open source middle-ware. The middle-ware focuses on problems typical to urban planning and therefore features a geo-data repository as well as a job runtime administration, to coordinate simulation models and its multiple views. The described system architecture is accompanied by two exemplary use cases, that have been implemented to test and further develop our concepts and implementations.

Keywords Middle-ware · Design-simulation-loop · Multiple comparative views · Distributed computing · Computational urban planning

1 Introduction

Among many new developments in urban planning, from a technical perspective it is foremost the increasing amount of data from simulation of urban phenomena that render the process of urban planning increasingly complex. This process often encompasses a wide variety of tools ranging from data acquisition, urban modelling, interactive simulations and 3d visualization. Integrating these tools to close the design-evaluation cycle is still in reality doomed by an integration through data files and manual format conversions.

Design-evaluation cycles require nearly real-time, complementary views on a planning site [1]. To not only create such views but also coordinate and synchronize their content as well as the corresponding calculation processes, we envision a design-simulation pipeline where highly specialized tools interact with each other to perform complex tasks. The benefit of such a pipeline for urban planning raises with the number of potential tools that can be coordinated with each other. This pipe-line is usually achieved with a middleware that assembles and coordinates software components through a messaging system. As an additional layer of abstraction a middleware has several advantages like easing the software development, establishing data standards and easing software maintenance. Individual components can be exchanged on the fly without a requirement for managing new software releases. This fits the volatile urban planning requirements that vary a lot from one project to another. Moreover, the middleware approach covers well the increasing demand of scalable computing solutions to verify or calibrate complex urban models with big data. As standard middleware solutions impose a rather strict design process that contrast the diverging requirements in urban planning, we decided to implement our own middleware solution that we call Lightweight Urban Computation Interchange (LUCI).

With the development of LUCI we followed two main questions: (1) How can we couple different simulations written in various programming languages to provide comprehensive information for urban design variants. (2) How can we test the functioning and performance of LUCI to proof it's suitability.

This paper was revised from the paper initially presented in FOSS4G Seoul 2015 conference.

✉ Lukas Treyer
treyer@arch.ethz.ch

[1] Department of Architecture, ETH Zurich, Wolfgang-Pauli-Str. 27, 8093 Zurich, Switzerland

[2] Future Cities Laboratory, Singapore-ETH Centre, 1 CREATE Way, #06-01 CREATE Tower, Singapore 138602, Singapore

Furthermore, the aim of LUCI is to simplify the middleware approach to a stage where planners are able to install and run it at the ease of any other desktop application. We think this simplicity is necessary to enable architects, planners and stakeholders to plan smart cities or smart neighborhoods, which in turn potentially fosters a bottom up and participation culture—an ever-growing topic in (western) modern urban planning. In the language of GIS this means we are looking for a way to bring the advantages of a network oriented GIS server to the desktop. This should enable users to (a) run simulations nearly in real-time for a planning workshop on many machines distributed in a local network as shown in Fig. 1 or (b) to communicate with citizens over the web or to (c) collect citizens feedback or knowledge for design tasks.

After a brief introduction about urban planning challenges, we outline recent trends in middleware development focusing on urban planning aspects. In Sect. 3 we will present the layered architecture of the LUCI middleware, its key functional features and provide additional information about its Desktop, Web and data interface. After this technical part we describe in Sect. 4 two cases in which LUCI has been used and tested already. The use cases range from a multi-screen environment in which we equipped an interactive planning workshop scenario with not only complementary but coordinated views, to web applications that communicate to LUCI with web sockets.

2 Related work

There is no best middleware solution as they are typically designed for specific purposes, targeting problems they were supposed to solve. Urban planning tasks usually involve several disciplines and require the processing of relatively complex models in the background. Off-the-shelf middleware solutions [2] therefore are of limit use, as they do not provide support for adequate data formats or planning functions. As we will present later in this chapter, Gorton's concerns about the middleware solutions back in 2003 were shared by researchers as for instance developers at TNO in the Netherlands. The urban planning scenario we want to support asks for more focus on qualitative and interactive aspects than the classical GIS suites offer. Those suites don't question the built environment as it is, but focus on recording, mapping, analysing it.

The two dominant integration styles in urban planning software are (a) sequentially forming a digital workflow chain or (b) following publish-subscribe model in a parallel way, where events are instantly propagated to all interested parties.

First research on digital workflow chains for urban planning have been conducted already for several years. They follow a similar principle in which collaborating tools are connected through Internet networks finally forming a seamless integrated urban planning pipeline. For instance, data acquisition, urban modelling, simulation and visualization tools can be linked so that a completely interactive exploration of design alternative becomes a reality. As Janssen et al. [3] mentions, this integration has to happen in a user friendly and flexible way to make this useful technology available to wider range of urban planners. He proposes a workflow configuration engine to manage tool collaborations within an urban planning pipeline (Fig. 2). Achieving a close design-evaluation cycle is especially important in design workshops as value raises with the capability to test proposed design solutions within a given workshop session. Since such proprietary tools rarely support the same data standard this middleware category often comes with data transformers. Programmable mapping functions can be used to manage the data translation process. The reality however shows that this task can become very complex and time consuming as data structures often may miss specific entries, have inconsistent columns or simply divergent conventions across sources.

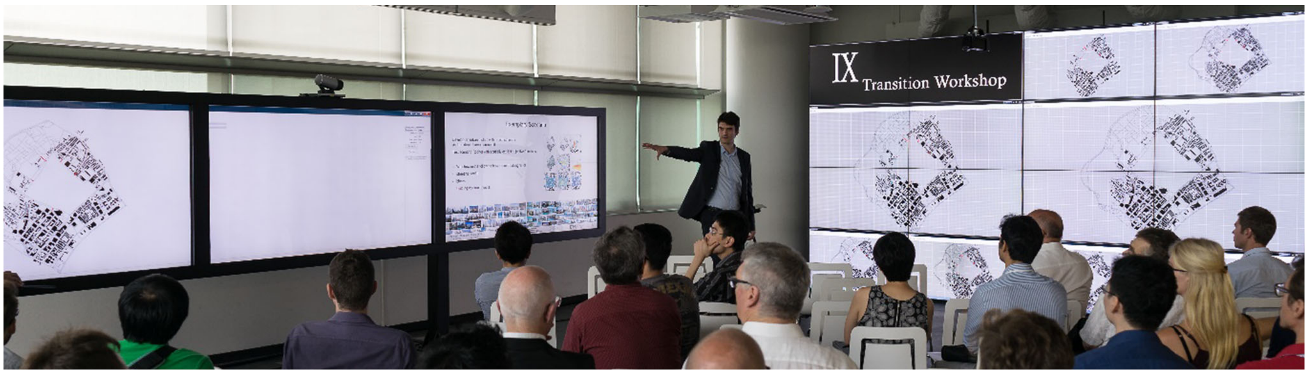


Fig. 1 Transition workshop at FCL in Singapore in April 2015 showing several views of the CPlan tool backed by LUCI imitating a planning workshop

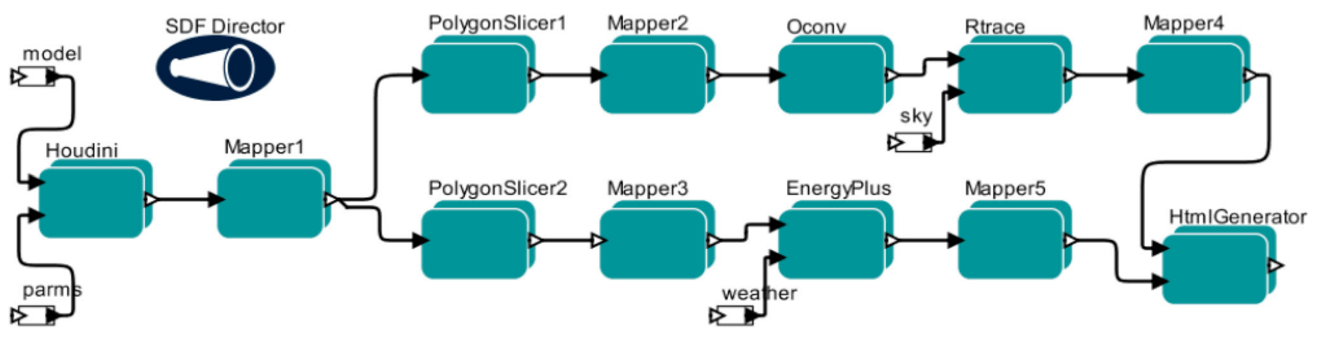


Fig. 2 Exemplary workflow in Kepler [3]

Publish subscribe systems enable the asynchronous linking of urban models and tools leading to richer view on the entire urban planning scenario. A good example is the Urban Strategy [4] framework from Toegepast Natuurwetenschappelijk Onderzoek (TNO) in the Netherlands. Urban Strategy accelerates and improves urban spatial planning by making information from linked, high-tech computer models available interactively [5]. The models simulate traffic, air quality, noise, external safety, sustainability, groundwater, costs and other aspects of our physical surroundings. Urban Strategy links a central database containing data on the built environment (derived from local authority datasets) to independent computer models (Fig. 3). The exchange between these models occurs via a separate communication layer. This enables changes in the results of the models to be immediately calculated into the other models. Changes can be made to the database containing data on the built environment: roads can be closed, houses built, transport routes altered, etc. The effect of these changes is calculated nearly real-time and made visible in Urban Strategy.

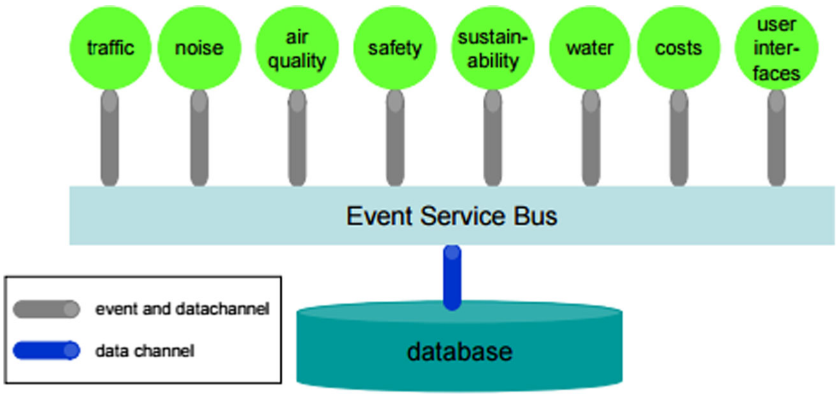
It is important to note that smart city platforms are often offered as a cloud computing solution [6]. Therefore we complement our LUCI middle-ware by providing sensor data (participatory sensing in real-time) and give access to historic government data. In return LUCI may provide cognitive computing models to achieve liveable cities that consider the human factor apart from the technology.

3 Middleware architecture

As shown in the related work section, there are quite a few middleware systems focusing on urban planning [3, 7–9]. We introduce an open middleware for urban planning that includes a geo-data repository and an asynchronous job management engine that allows a supervised execution of modeling, simulation and visualization tasks in a network. Our solution focuses on simple configurability and usability, parallel computation of urban simulations and flexible web integration. In contrast to existing solutions LUCI is platform independent, open source and allows to encapsulate complex simulation tasks for urban planning in a straight-forward manner that relates more to tools that support creative design tasks compared to typical GIS tools.

It is implemented combining the Message Broker model with a simple Server-Client architecture. We use MQTT for notification and a separate TCP socket for the content ex-change. Similar to MQTT the content exchange sockets remain open as long as the client is connected. Content messages follow a fairly simple protocol, in which messages consists of a JSON header followed by binary data if needed. The JSON header must contain one of the three keywords “action”, “result”, “error”, where action and error are strings and result a JSON object.

Fig. 3 Software architecture of Urban Strategy [5]



3.1 Terminology

The name "action" corresponds to a terminology we define in LUCI as follows. "Actions" are small pieces of code used for the administration of data, simulation and data conversion tasks. They are supposed to run very fast and when called from a client they run synchronously. The client only gets an answer once the action is finished. "Services" on the other hand is the name for asynchronous tasks supposed to run for a longer time that would not be conceived as responsive by a user. "Convertors" are a class in-between running synchronously at the moment but being designed to run asynchronously with only a few modifications. "Actions", "Services" and "Converters" are part of a plugin system, i.e. dynamically (re-) loaded at runtime. Thus a restart of LUCI that would reset all socket connections is not necessary. As shown in Fig. 4 there is also a category for "Database" plugins. Even though not reloadable at runtime this allows LUCI to have different database adaptors, which in turn enables users to work with their preferred database. Apart from the plugin structure, Fig. 4 also shows the basic idea of the data structure. We map service in- and outputs to separate tables. "Scenarios" denote the main unit in which the geo-data repository is divided into. They are shared among the services. For more information on the data structure refer to Sect. 3.4.

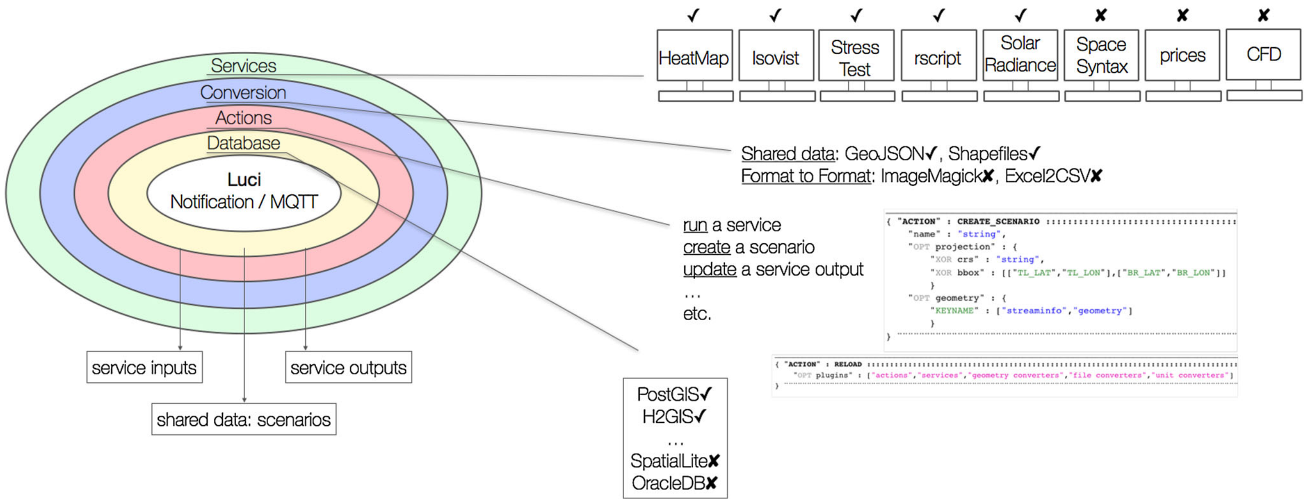
3.2 Graphical user interface(s)

Administrative controls including an interactive console are available from a Desktop System Tray Menu as shown in Fig. 3. Nevertheless, LUCI can also be configured to run headless without any GUI. By now, the most important control available from this menu is the interactive console. It is intended for developers to send actions to LUCI as raw JSON strings most likely for testing reasons.

Other commands available from that menu are Starting and Stopping LUCI, open a PDF documentation (Open Specification) and a few more, as listed in Fig. 5.

3.3 Web interface

A major highlight of the LUCI middleware is the fact that it can be embedded in any web system through web



```
{ "ACTION" : CREATE_SCENARIO ::::::::::::::::::::::::::::::::::
    "name" : "string",
    "OPT projection" : {
        "XOR crs" : "string",
        "XOR bbox" : [["TL_LAT","TL_LON"],["BR_LAT","BR_LON"]]
        }
    "OPT geometry" : {
        "KEYNAME" : ["streaminfo","geometry"]
        }
}
```

```
{ "ACTION" : RELOAD ::::::::::::::::::::::::::::::::::::::::::::::::::
   "OPT plugins" : ["actions","services","geometry converters","file converters","unit converters"]
}
```

Fig. 4 LUCI's pluggable architecture and data structure

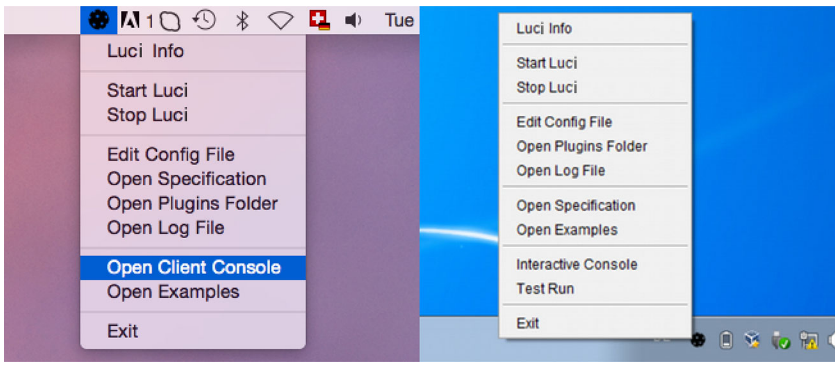


Fig. 5 Desktop system tray menu (OSX/Win7)

sockets. This opens the door for a wide variety of HTML5 web applications. We use ActiveMQ as a MQTT broker (see Sect. 3.8). Upon starting up the broker, it also starts a jetty webserver to support MQTT over websockets. This jetty instance also serves LUCI's webcontent. In the future the web interface should develop into the main instance from which a user can administrate and monitor LUCI, its service instances and scenarios. Among others we envision a flow diagram to interactively visualize service instances and perhaps other parts of a LUCI scenario. As a showcase of LUCI's capabilities offered by websockets, please refer to Sect. 4.2 "Teaching the Unkown".

3.4 Data structure

LUCI is supposed to run database-agnostic. This is achieved through mostly standard SQL code and all the agnostic parts being part of a database specific plugin in the database layer (see Fig. 4). At the moment Postgres and H2 are supported. The data structure can be subdivided into two main topics: the inputs and outputs of the services as well as the geo-data repository, which in the perspective of the individual services could also be termed as "shared data".

The most important feature of the service related data is that we need to be able to relate every generated output to the corresponding input data. We solve this with timestamps: The services can operate on either only their input data or as well their input data plus shared data. Therefore, in an additional table we store the newest available timestamp from input table, shared geo-data and the newer out of the two in a third column, which at the same time also is being used as the identifier of a call to a service. This call-ID then is being used to identify the service outputs; the outputs table holds a column call-ID, which is the newest timestamp of its inputs.

The data structure of the shared data is driven by the goal to make query code as simple as possible and put the complexity into the insert and update code. As introduced in the terminology Sect. 3.1, the shared data is organized in scenarios. Among seven predefined columns a scenario is defined by its attributes and an optional coordinate system identifier. In other words a scenario defines a project space in which all geometry shares the same attributes. Besides self-explanatory attributes like "geomID", "geom", a general purpose "flag", "userID", "timestamp" a scenario holds:

- a "batchID": LUCI among other use cases is being used for an evolutionary optimization process in which many different variations of the same scenario are created and evaluated. To avoid mistake such variations with versions planned to be implemented in the future, we call those variations "batches".
- a "layer": Similar to layers in CAD applications a layer in LUCI is exclusive. So, geometry cannot be part of two layers.

For each scenario four tables are being created. Besides the main table holding the current state of a scenario, there is also a history table with a nearly identical structure as the main table. The only addition is a timestamp column to store the time at which a record from the main table has been deleted. The two remaining tables will be used for versioning in the future development of LUCI.

3.5 Remote services

Among a few "local" services that reside in plugins/services folder a major benefit of using LUCI results out of its network orientation. It features distributed computing, load balancing. "Remote" services are a key element in the implementation of the parallelization and/or distributed computation capabilities of LUCI. They are characterized by two main attributes: Firstly, from a client's perspective remote services are indistinguishable from local services when being called. Secondly, any client can register as a service.

Upon registration a service describes its inputs and outputs. The inputs of any future call to that service will be verified by LUCI using this input description. The input/output description is very similar to the capabilities of web process services (WPS). In the future we could even think of converting the description to WPS when exposing the available services to the web.

Since the data is extensively being marked with timestamps, LUCI is able to send only the updates in the scenario data and input parameters since the last execution. Therefore the "get_scenario" and "get_service_inputs" actions both support the concept of time-range-selections, which is basically a parameter of the action call consisting of one of the keywords "from", "before", "until", "after" and a timestamp as the value. To make use of this partial data extraction, the remote service must implement some sort of data cache to which the updates can be added.

3.6 Communication protocol

Messages in LUCI consist of a JSON header and optional binary attachments. The first 16 bytes of all messages encode the length of each header and attachments with an 8 bytes big endian number respectively. This is crucial since connections are not being closed, but remain open during a session (web sockets) or until the connection gets closed by either client or server (TCP/IP). The attachments part can contain multiple byte arrays. All of them must be described in the JSON header by a streaminfo object; a JSON object with predefined structure and keywords. If processing of the header fails, using the informaiton oft he first 16 bytes, all subsequent bytes can still be read, which clears the socket for the next message.

At the moment messages can be sent through TCP/IP and web sockets. Parallel messages are not allowed, so each message must be answered before the next message can be sent. This shifts the complexity of parallelization away from the client to LUCI and the term call ID remains free for services. As mentioned in the terminology Sect. 3.1, we distinguish between actions and services. Actions are similar to remote procedure calls with the exception of not having a call ID. Messages always call an action by using the "action" keyword. Any message in LUCI must either contain one of the keywords "action", "error", "result", which "action" and "error" holding a string-value, and "result" a json object.

Actions themselves are plugins similar to local services, database adapters, or data converters. LUCI comes with a standard set of actions, which can be extended or adapted to the specific needs of a project, just as services can. In Sect. 4.2 we show an example of how LUCI can be adapted to special needs by implementing dedicated services and/or adapting actions and converters.

3.7 Converters

Converters are plugins that call predefined functions of database adapters to store geometry in the scenario table. Supported formats so far are:

- GeoJSON
- Shapefile
- OSM-JSON (read-only)

DXF and other formats closer related to CAD are on the task list for future development. Converters must not only translate the information from one format to the database, but also implement a few features specific to LUCI:

- Attribute Mapping: a JSON object being part of the stream info object that tells the converter which attribute (e.g. ID) should be mapped to the seven predefined attributes (e.g. geomID) described in the data structure section.
- Delete_list: a property being stored in the format itself that tells the converter which elements should be deleted from the table, i.e. moved to the history table. E.g. in GeoJSON the delete_list is a property of a feature that holds no geometry.

3.8 Job management/MQTT

Jobs, in LUCI being called service instance, can be run synchronously or asynchronously. In case it should be run asynchronously the service instance must be created first in order to retrieve a service instance ID (SObjID). As discussed in the data structure Sect. 3.4 services can have inputs and outputs, which they define at runtime. Upon instance creation all input parameters of a service instance are being stored to the data-base. Whenever the service is being run, its inputs are loaded from the database. In theory the service can be re-run as many times as desired. Still, the service can store the outputs that belong to one single call ID (see Sect. 3.4) only once. Since the call ID is always equal to the newest timestamp of services inputs re-running services only makes sense, if one of its input parameters has changed.

To listen for such changes we use the Message Queue Transport Transfer (MQTT) protocol, a publish-subscribe framework. It was developed by IBM, is open source and builds on top of TCP/IP and web sockets. It is referred to as the protocol for the Internet of things. A LUCI service instance can either subscribe one of its inputs to the output of another service instance or subscribe the instance as a whole to the termination of other services, which will cause the instance to run immediately after another service instance has finished. With this setup service instances can be represented in a flow diagram, which is the intention of the configuration interface mentioned in Sect. 3.3. Using MQTT enables client applications to run previously created service instances simultaneously with one publication to MQTT. Furthermore, it enables them to monitor all service instance related activity.

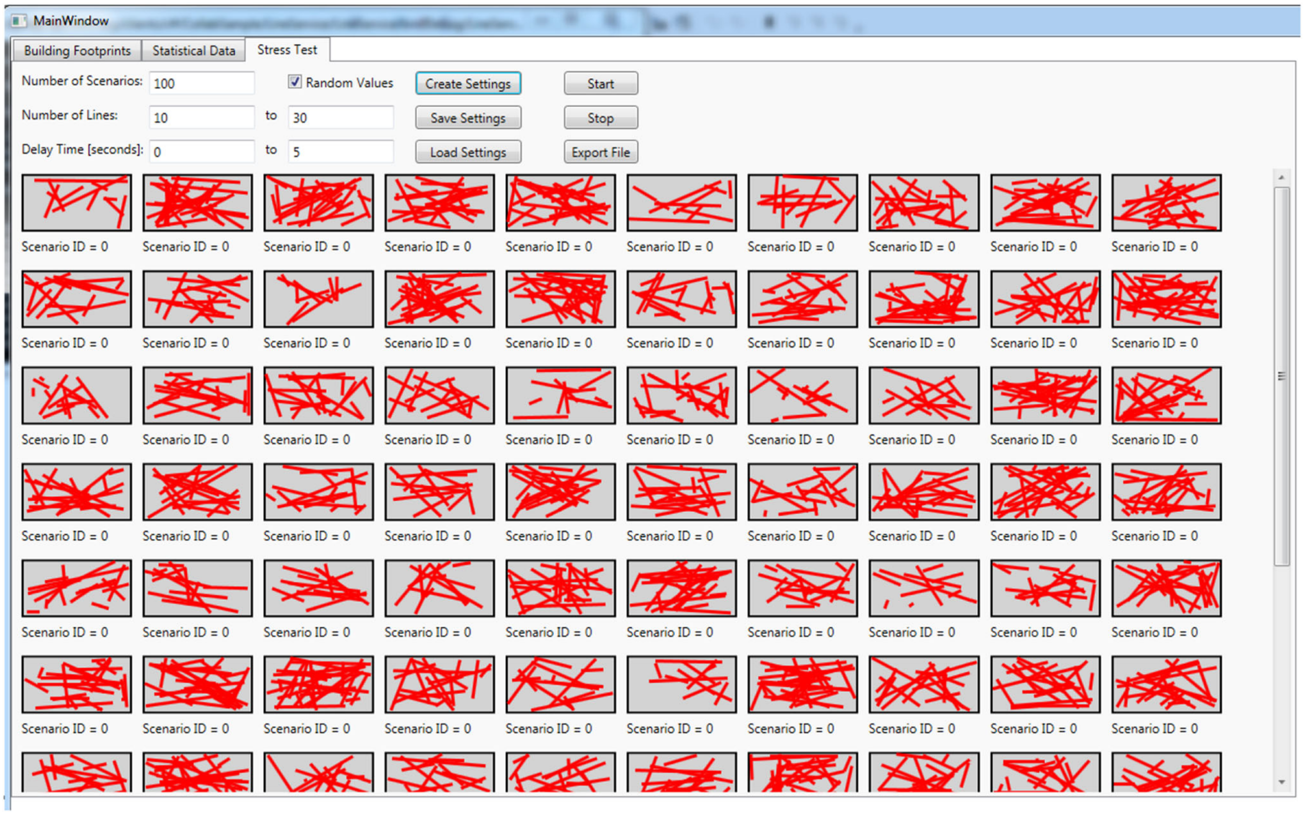
Synchronous calls cannot be called through MQTT, but they must be called through the "run" action built-into LUCI. All service inputs and outputs are not transferred to the data-base but directly the (remote) service and back to the client. The run action will wait until the service completes.

3.9 Accessibility and availability

LUCI is available open source (MIT license) on bitbucket.org [10]. It comes with connection libraries in Java, C#, Python and Javascript. With these libraries client development is being simplified. It is mainly the communication protocol that is being implemented by these libraries which then allow users to connect to LUCI using simple API's in the above mentioned respective languages. The main motivation to provide these libraries is to abstract attachment handling (Java, Python, Javascript) including the respective JSON header (C#) away from API users.

4 Dynamic load balancing experiments

The mere existence of the above mentioned connection libraries lead to not use load testing tools, since by using the connection libraries for testing we were able to test both: LUCI and connection library.



Number of Scenarios: 100 ☑ Random Values
Number of Lines: 10 to 30
Delay Time [seconds]: 0 to 5

Fig. 6 User interface for the line service. Researchers can specify the number of generation steps with the maximum scenario number and the scenario complexity by specifying the *number of lines* and delay time

The benefit of having a middleware is the possibility to distribute the computational effort on different computers to the price of having additional messaging effort to coordinate the processes and join the final results before they are send back to the urban planning application that has issued these services.

In the following we want to gain an in-depth understanding of how the entire system behaves under heavy load conditions in order to refine its configuration or even make changes to the middleware implementation.

4.1 Methodology

We consider three experiments. First we want measure the realistic messaging overhead by executing the computational services first as embedded service part of an urban planning software and later remotely executed on a separate machine. In the second experiment we want to gain a better understanding about the parallelization benefit by running the service with a fixed number of services on the server one after another and later execute service requests elastically based on the demand situation of the client (auto-scaling scenario). Both experiments measure a technology's performance under normal conditions. For our final performance experiment, we want to observe what impact increasing request rates have on the mean response time within the LUCI network leading to heavy transaction loads.

4.2 Design task generation

We developed a line graph generator to automate load balancing and network-re-configuration experiments with different scenarios. The line graph generator consists of a client and server application. It emulates the human producing simple geographic layouts and to the same time executes a given workload on the middleware server for the experiments. Users can specify a workload scenario for an experiment with the user interface of the line graph generator shown in Fig. 6. The workload specification consists of service descriptions that contain following information:

- The total number of scenarios to generate in a random or in a predefined order.
- The range of the number of lines and the correlated delay time reflecting the computational complexity of the assumed design task.

For the experiment, we randomly generated design tasks with a given 10–30 lines with a delay time of up to 5 s.

Figure 7 shows the resulting distribution of delay times for each design task.

4.3 Scenario based event logging

After the drawing task generation has been finished two experiments are executed to verify the load balancing capabilities of the LUCI middleware. The services executed on the server-side specify the demands for respective urban planning methods in the test implementation. The task is simulated by painting a given number of lines and together with an additional waiting time simulating the method implementation e.g. triggering an external entity, for instance a query to a remote GIS repository. In addition, different load balancing options can be chosen to verify its impact.

During the experiments, basic performance measurements are obtained through an event logging tool part of the line graph generator. Timestamps are recorded and stored in CSV files during the service request issued by the client, job queuing at the LUCI middleware and for the service execution at the server side. The LUCI experiment report produced by the post-processing scripts includes the following information:

- The description of the client and server scenarios, used by the workload generation.
- The number of created services and additional load balancing information
- The average queue length for the middleware broker
- Time stamps for scheduling, start-, and end time of the service in HH:MM:SS
- The mean completion time, mean messaging time, mean service duration
- Final execution state (with or without error).

4.4 Result evaluation

For the first experiment we specify different line services with runtimes of 0, 5 and 15 s drawing 10 lines. The servides are executed first locally on a computer and later through a remote computer in the LUCI network, where services are run on various computers. The computation services executed on the remote machine then calculated the total line length and waited for the remaining runtime before the results were sent back to the client. The results are shown in Table 1. The table shows that services with a runtime above 10 s benefit from their parallel execution, below that the messaging effort reduces its benefit.

Based on the first experiment we plotted a time series sheet for service requests executed with a predefined number of services on the remote machine and later through elastic service creation approach. The latter scenario starts services automatically on the server and kills it after it has performed its task and has reached a maximum idle time. In Fig. 7a all services are executed in a series of 10 services one by one. This of course has lead to a stepwise increase of the overall execution time. The services in Fig. 7b are started dynamically on demand keeping the overall execution time of each service almost constant but to the price that the RAM memory gets filled up with the time. In a realistic scenario it is therefore necessary to set an upper limit to avoid memory problems.

For the last experiment we stepwise increased the number of services up to 1000 services and observed a quite robust scaling behavior shown in Fig. 8. Under such a heavy load condition it turned out that running 100 services in parallel is enough to balance the overall execution load by replacing consistently old with new services. The peaks in the beginning stem from the stepwise increase of parallel running services—roughly about 10 additional ones every 30 s—till the demand became finally balanced again with the number of freeing up services. Except few exceptions the overall response time was within close boundary proofing that the amount of messages sent to coordinate the LUCI network scales well.

5 Use cases

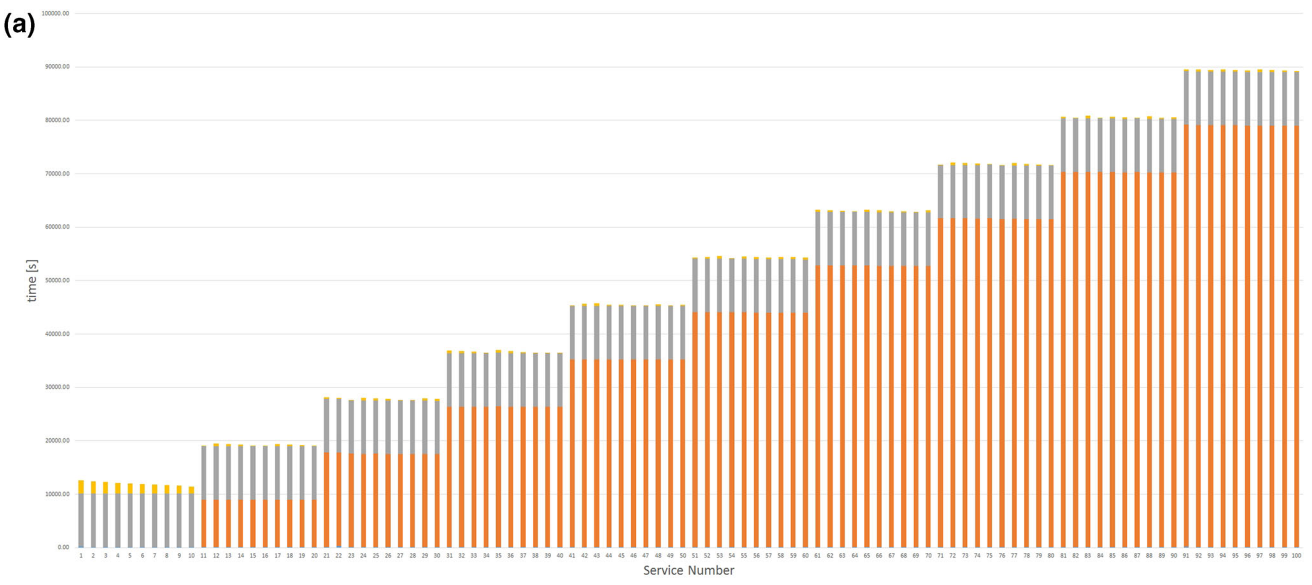
5.1 Transition workshop, Singapore

Equipped with an early prototype of LUCI we conducted a workshop called “Design Space Exploration for Urban Compaction” at the SmartGeometry conference held at Chinese University Hong Kong in 2014 [11]. The concept of multiple coordinated views [1] representing the coordinated parallel simulations was very well received. As shown in Fig. 9 we refined this concept for the transition workshop of the Future Cities Laboratory of the Singapore ETH Centre in April 2015. We were able to present a system with synchronized views across several machines in the network using MQTT’s publish-subscriber mechanism.

In the context of the transition workshop we assessed the applicability of the LUCI system by using an example scenario in the district Rochor in Singapore. To prove that our approach works in an existing urban context we assume that this district needs to be completely re-planned. This exemplary area in Asia emphasizes the urgent need for fast and comprehensive planning systems. Necessary data on the street network was taken from Open Street Map, and information about neighboring built structures in 3D was available from the Future Cities Laboratory of the Singapore ETH Centre.

We used the LUCI system in combination with an evolutionary multi-criteria optimization method (EMO) for

(a)



(b)

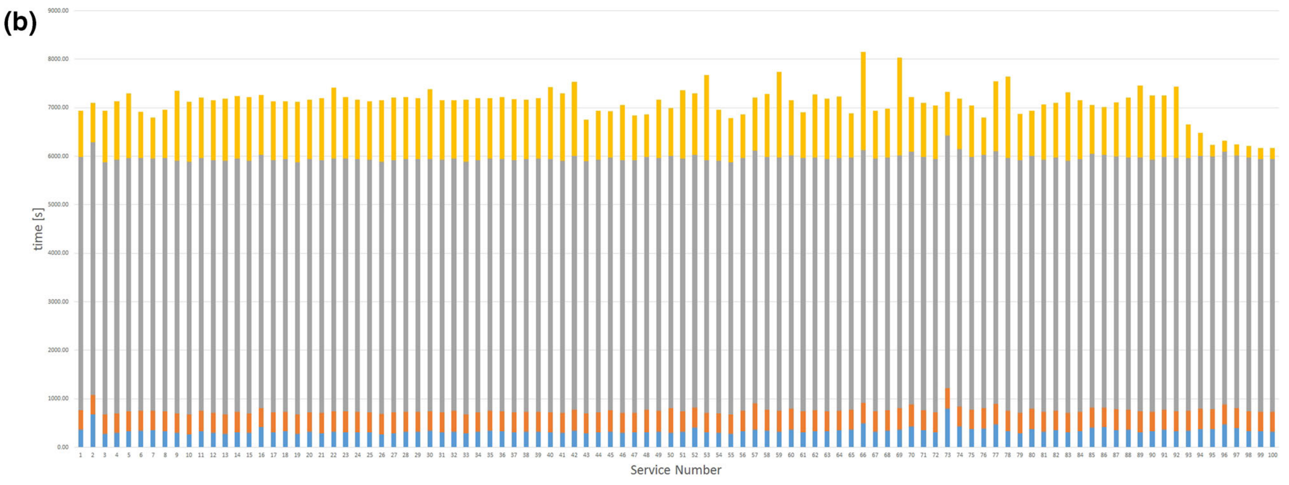


Fig. 7 Load balancing scenario with 100 services in **a** *top*: static amount of 10 processes and **b** *bottom*: elastic on-demand service generation. The colors provide time information for (i) *blue* scenario generation, (ii) *orange* service scheduling, (iii) *grey* service execution and (iv) *orange* service response. (Color figure online)

Table 1 Comparison of service response times in local and remote execution scenarios with different service runtime

Service runtime	Single computer (s)	Distributed computing
0	39	73
5	62	84
15	198	130

creating road networks with defined centrality characteristics, such as integration or choice for defined locations [12]. Inside the blocks of the road networks we generate building layouts with defined densities, taking into account specific properties of the open space qualities measured by Isovist fields [13]. The evaluation mechanisms were coupled as services with the EMO. Figure 10 illustrates how the resulting planning system can be used to support an urban planning process (Fig. 11).

This kind of computational planning process we call evidence-based planning. It helps the designer not only to eliminate potentially problematic configurations but also to meet the original design requirements.

5.2 Teaching the unknown, Zurich

"Teaching the Unknown" is the title of a contribution we made to an exhibition by Moon Kyungwon & Jeon Joonho at the "Migros Museum für Gegenwartskunst" in Zurich from August 29th until November 8th 2015. The title of the exhibition was "News from Nowhere: Zurich Laboratory". The art piece consists of a touch screen table that shows a website running on LUCI. The website shows the images

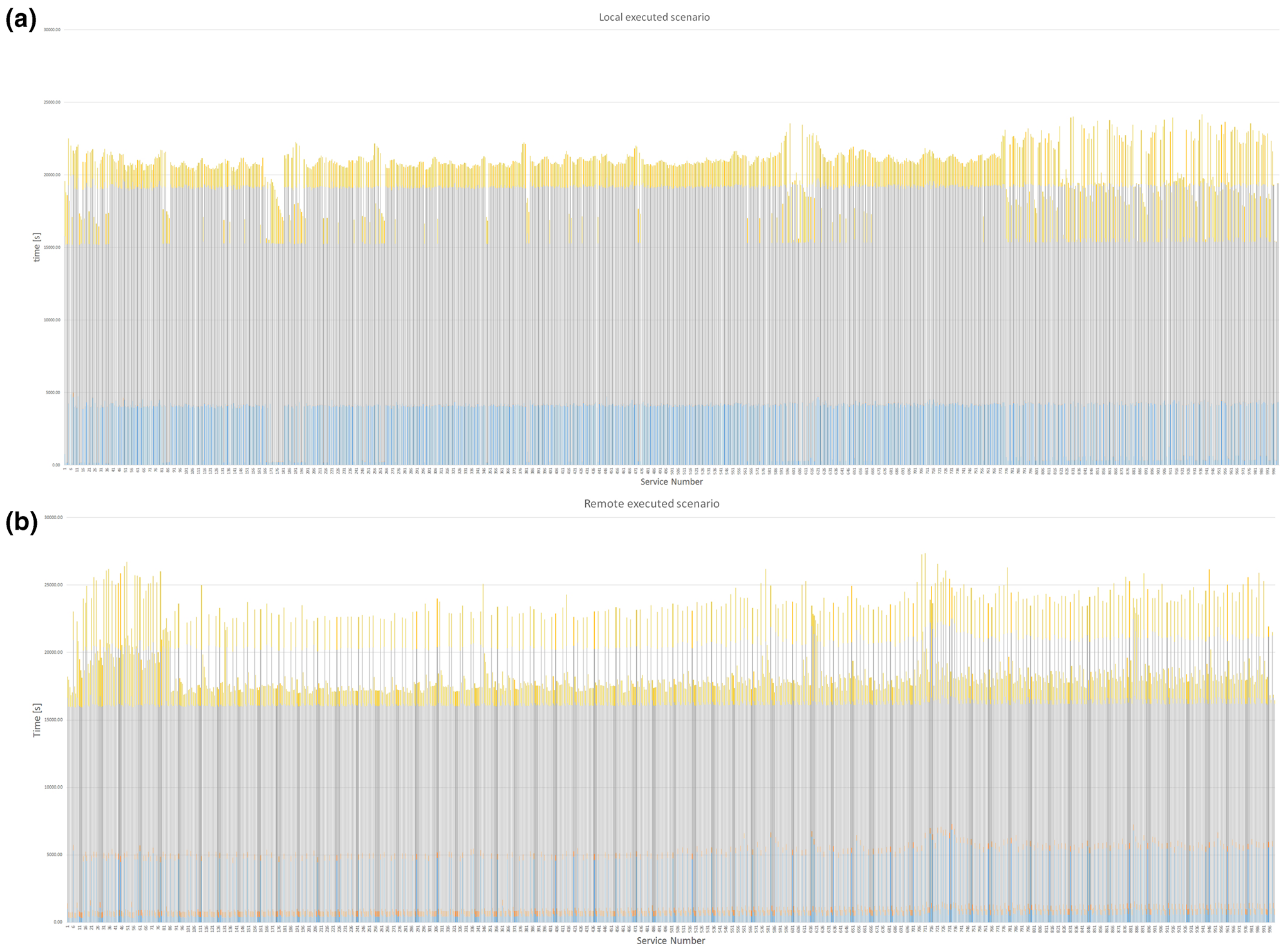


Fig. 8 Load balancing scenario with 1000 services in (**a**) local and (**b**) remote executed scenario. The colors provide time information for (i) *blue* scenario generation, (ii) *orange* service scheduling, (iii) *grey* service execution and (iv) *orange* service response. (Color figure online)

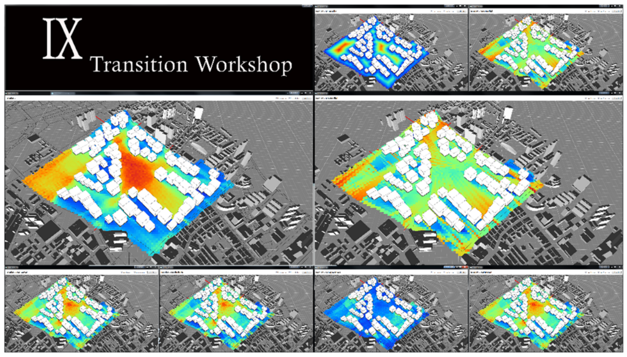
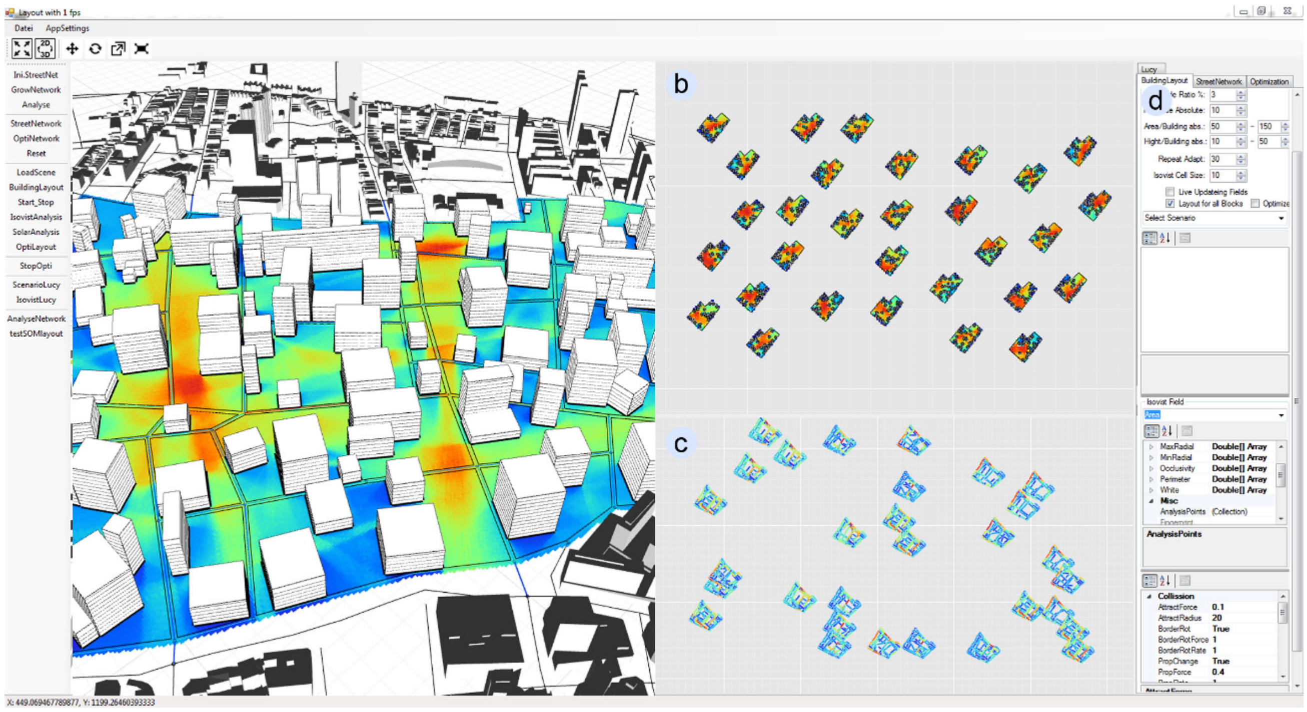


Fig. 9 Coordinated and multiple views on multiple screens (machines)

that were handed in by roughly 1000 students as a part of their exercises in a Massive Open Online Course (MOOC) on future cities held by Prof. Dr. Schmitt [14].

The website uses a map from Mapbox being semi-transparently laid over an image. The location where an image was taken is marked and an additional text box showing the description to the image. In an additional gaming mode visitors must select the city in which a randomly chosen image was taken. The last mode shows the rank of the player and the cities.

The students sent in 2083 images. For the game we selected 240 images from 120 cities. In LUCI each image is stored as a point with the city name, the image, the user name and the description being an attribute of that point. In order to speed up the client-side script (javascript), we wrote a small LUCI service that aggregates all selected images from the same city, so that in the Javascript we must not check for images with the same location. A second LUCI service ranks the cities based on the number of correct guesses. The player score is put together by the average distance to the correct city markers from the markers chosen by the player. A third LUCI service counts how many players achieved a better score, which is then used to display the rank of the player in the overall player history of the exhibition.



Ratio %: 3
Absolute: 10
Area/Building abs.: 50 - 150
Hight/Building abs.: 10 - 50
Repeat Adapt: 30
Isovist Cell Size: 10
[ ] Live Updateing Fields
[✓] Layout for all Blocks
[ ] Optimize

MaxRadial	Double[] Array
MinRadial	Double[] Array
Occlusivity	Double[] Array
Perimeter	Double[] Array
White	Double[] Array
Misc	
AnalysisPoints	(Collection)

Collission	
AttractForce	0.1
AttractRadius	20
BorderRot	True
BorderRotForce	1
BorderRotRate	1
PropChange	True
PropForce	0.4

Fig. 10 EMO-Software prototype showing the main areas of the user interface: **a** a 3D view combines one solution out of each archive, design solutions of the archives for **b** buildings layouts and **c** street networks, and **d** fields for the user input of size of population, number of generations, etc

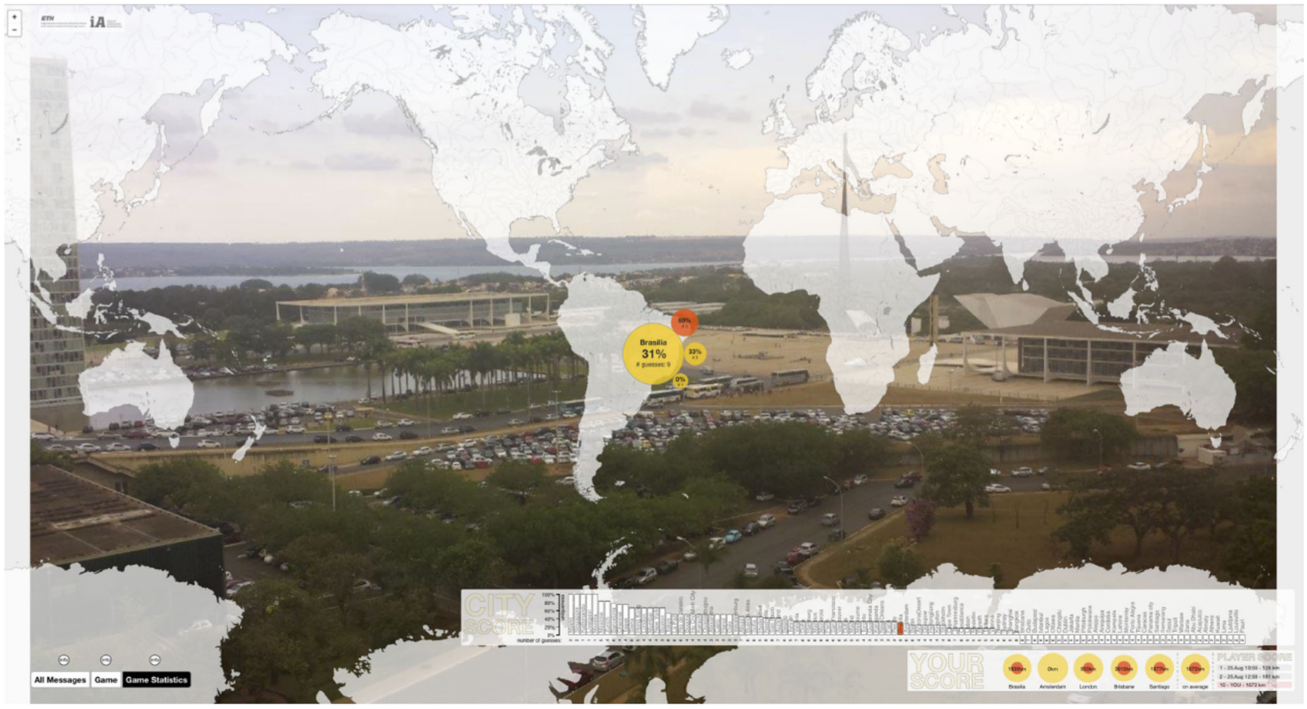


Fig. 11 Screenshot of the game statistics mode showing city and player score

While this project was mainly used to stabilize LUCI's data structure, one could think of LUCI's job management capabilities through which services could run automatically after a students upload more images in a future MOOC.

6 Conclusion and outlook

With the presented concept of LUCI as described in Sect. 3 we are able to answer our initial question, how can we couple different simulations that are written in various programming languages and that can run on different platforms. The second question about the functioning and performance of LUCI was positively answered by the dynamic load balancing experiments in Sect. 4 as well as by the use cases presented in Sect. 5.

For the development of LUCI we focus on small groups of planners or researchers that want to collaborate with each other and perhaps scale their collaboration to a larger group of participants at a later stage. As mentioned, with such a focus the simplification of the middleware approach becomes crucial. Simplification towards a desktop application for us means to avoid as many enterprise frameworks as possible that would hinder potential users to quickly develop a custom action, service or converter. We provide simple Plain Old Java Objects (POJO)-like access to LUCI that should allow users such as planners or researchers to easily adapt LUCI to their needs in a transparent way. The integration of an H2 database adaptor takes away the hassle of installing a PostGIS database for people unfamiliar with GIS databases such as architects or planners.

In the future we plan to integrate LUCI with the "Tool Library", a solution that makes view and tool configurations in a multi-screen situation room accessible and controllable from a web-based application. In this regard we would also like to create an editor that allows users to put together service instances in a flow diagram. In this respect we would like to mention LUCI2 [15], a refactoring we started to work on to consolidate the concepts described in this article. Furthermore, we could think of exposing LUCI's service descriptions to the web following the web processing service (WPS) standard as well as adding support for DXF and other NURB-based geometry formats. We also would like to add support for transformation matrices, so that the communication with regular 3D editors could be streamlined, since not the whole geometry would need to be sent through the network, but only its transformations. And as mentioned in the data structure section, the data structure has been designed with versioning in mind. A future implementation of this will extend LUCI's capabilities even more.

Acknowledgments We would like to thank Daniel Zünd, Alessandro Forino, Artem Chirkin, and Patrick Janssen for the many inspiring discussions and helpful ideas on LUCI.

References

1. Baldonado, M. Q., Woodruff, A., & Kuchinsky, A. (2000). Guidelines for using multiple views in information visualization. *Workshop on advanced visual interfaces (AVI'00)*.
2. Gorton, I., Liu, A., & Brebner, P. (2003). Rigorous evaluation of COTS middleware technology. *Computer, 36*, 50–55 + 54–55. doi:10.1109/MC.2003.1185217.
3. Janssen, P., Stouffs, R., Chaszar, A., Boeykens, S., & Toth, B. (2015). Custom Digital workflows with user-defined data transformations via property graphs. *Design Computing and Cognition'14* (pp. 511–528). New York: Springer.
4. Schelling, A., Schrijver, J., & Wilmink, I. (2007). Urban Strategy, een nieuw instrument voor ruimtelijke planvorming. *Colloquium Vervoersplanologisch Speurwerk* (Vol. 22).
5. Hofman, W., Lohman, W., & Schelling, A. (2011). A flexible IT infrastructure for integrated urban planning. *Journal of Theoretical and Applied Electronic Commerce Research, 6*, 16–25. doi:10.4067/S0718-18762011000100003.
6. da Silva, W. M., Alvaro, A., Tomas, G. H. R. P., Afonso, R. a., Dias, K. L., & Garcia, V. C. (2013). Smart cities software architectures. *Proceedings of the 28th annual ACM symposium on applied computing - SAC '13*, 1722. doi:10.1145/2480362.2480688.
7. Al-Jaroodi, J., Mohamed, N., Jiang, H., & Swanson, D. (2003). Middleware infrastructure for parallel and distributed programming models in heterogeneous systems. *Parallel and Distributed Systems, IEEE Transactions on, 14*(11), 1100–1111.
8. Borst, H. C., Salomons, E. M., Lohman, W. J. A., Zhou, H., & Miedema, H. M. E. (2009). Urban strategy: Noise mapping in instrument for interactive spatial planning. *8th European conference on noise control 2009, EURONOISE 2009, 26 October 2009 through 28 October 2009, Edinburgh, PART 3* (Vol. 31, pp. 1–9).
9. Romualdo-Suzuki, L., Finkelstein, A., & Gann, D. (2013). A middleware framework for urban data management. In *Proceedings of the 2013 ACM conference on pervasive and ubiquitous computing adjunct publication* (pp. 1359–1362).
10. Treyer, L., Forino, A., Meixner, C., Chirkin, A., Klein, B., Koenig, R., et al. (2013). Luci Bitbucket Repository: http://bitbucket.org/treyerl/luci.
11. Koenig, R., Standfest, M., Treyer, L., Wei, Z., & Klein, B. (2014). Design Space Exploration for Urban Compaction: http://smartgeometry.org/?option=com_content&view=article&id=259&Itemid=201.
12. Koenig, R., Treyer, L., & Schmitt, G. (2013). Graphical smalltalk with my optimization system for urban planning tasks. In R. Stouffs, & S. Sariyildiz (Eds.), *Proceedings of the 31st eCAADe conference - volume 2* (pp. 195–203).
13. König, R., Standfest, M., & Schmitt, G. (2014). Evolutionary multi-criteria optimization for building layout planning-exemplary application based on the PSSA framework. In Proceedings of the 32. International conference on education and reserach in computer aided architectural design in Europe.
14. Schmitt, G., Hebel, D., Koenig, R., Klein, B., Brülisauer, M., Tapias, E., et al. (2014). Future Cities. https://www.edx.org/course/future-cities-ethx-fc-01x-2.
15. Treyer, L., Zuend, D., Forino, A., & Chirkin, A. (2016). Luci2 Bitbucket Repository: http://bitbucket.org/treyerl/luci2.