Architecture of cloud execution platform for system dynamics models

Alexey Mulyukin[0000-0001-5241-5388] and Ivan Perl[0000-0002-8903-405X]

ITMO University, Saint-Petersburg, Russia
alexprey@yandex.ru, ivan.perl@corp.ifmo.ru

Abstract. Nowadays computer modeling area is very popular and mostly researches interest in cloud computing in computer modeling rapidly growing. In scientific world math models with high complexity are continuously developed in different areas of applications, that is a cause of this growth. In this way our team should quick respond for new user requirements and growing of computation complexity for system dynamics models. Previously software architecture of sdCloud platform was complex and hard to maintain and extend. The new architecture based on micro-services infrastructure and Enterprise Service bus provide flexibility, scalability and reliability of computation platform. In this paper we describe all details of software architecture that was we built for sdCloud platform to compute system dynamics models. The paper paid special attention to the communication process of services with each other using the Enterprise Service Bus and introduce a new term – services responsibilities zone.

Keywords: Software Engineering, Enterprise Service Bus, Cloud Computing, System Dynamics, Computer Modelling.

1 Introduction

Nowadays computer modeling area is very popular and mostly researches interest in cloud computing in computer modeling rapidly growing. In scientific world math models with high complexity are continuously developed in different areas of applications, that is a cause of this growth. System dynamics is an aspect of systems theory which is an approach to understand the dynamic behavior of complex systems. The system dynamic models consist of stocks and flows. The stocks in scope of system dynamic represent some real values of our world that can be changed in over time. The flows in scope of system dynamic represent a function describing stocks values changes. Those simple elements allow constructing models of any complexity level. Such models can describe

CEUR-WS.org/Vol-2344/paper4.pdf

real world processes or systems in required scale for specific research needs. The system dynamic models cover many areas of our world including but not limited to economic, medicine, social, mechanic and engineering. In scope of system dynamics used the system of linear equations that solved by iterative approach in required model time space range that defined by researches that investigate model behavior. The model solving process is also can be called as model execution process. When we are talking about execution of a system dynamics model, we are assuming a process of sequential computation of model states over a given period with the provided modeling step, representing a minimal time frame to navigate in modeling results [1]. For example, we can take the simple epidemic model, that describe how disease spreads among area population, and this model built for answering the next question: "Can disease spreads to all population or disease is disappeared and all population will be healthy?" and processes of answering to this question called is model execution [1], [2], [3].

To execute system dynamic models required specified tools. Nowadays presented many different tools for researchers. Most of those tools are open source and supports different model formats, like xmile, vensim, stella, etc. and each tool implemented in its own way with different technologies. For example, looks at PySD and SDEverywhere open source libraries [4], [5]. The PySD library uses a python environment to handle system dynamic models and execute it; at the same time the sdEverywhere library wrote on JavaScript to handle models and then use C language to execute models. Meanwhile, we also can use different types of computing systems to optimize execution performance, for example we can move execution process from regular CPU with x86 architecture to GPU [6] or different types of CPU, like an Elbrus with VLIW architecture etc.

2 Overview of current platform implementation

The sdCloud – is a cloud platform for working with system dynamics models [7], [8]. The key feature of this solution is simple user interface which accessible from any modern web browsers. sdCloud allows to users create and execute models and then make analytics for given modelling results.

For the first time sdCloud solution was built on few complex applications: Web UI & API, executor service, Python bridge. Communication between those parts was implemented by different ways, some parts communicates via DB server queue, some parts communicates via HTTP channel; we use a system pipelines to direct communication with system processes created by sdEverywhere library. This approach really is chaotic and is not flexible to add something new with different technology stack, like a GPU model executor, that

wrote on C# and use OpenGL bridge to communicates with GPU on different platforms: windows or linux-like.

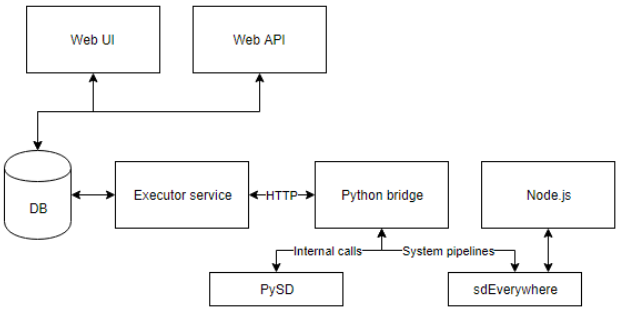

Fig. 1. The scheme of current architecture of sdCloud platform.

This architecture is complex, hard to maintain and have following issues:

- Hard to extend this architecture to add a new tool for model handling;
- Hard to scale solution to improve performance by horizontal scaling;
- Hard to add new integrations in our platform;
- Hard to implement complex solutions related with model and results data processing;
- Hard to understand code and dependencies between parts of code for new developers.

So, we should change this architecture and solve all issues described above. Our development team faced a challenge of finding an approach to build flexible, reliable and scalable solution for modelers. We should build an architecture that can seamlessly integrate existing tools to work with system dynamics models and provide an easy way to add different kinds of hardware to model compute in our platform.

For the beginning we mention following requirements for new architecture of cloud platform:

- High reliability for whole platform: we should provide a good solution for our users with minimal impact for them in case when something goes wrong in our internal infrastructure;
- Good flexibility of architecture: we should have easy way to add new complex features related with model execution and data analysis;
- Isolation of each tool with possibility in order to use libraries based on different technology stacks;

— Transparency: understandable code and architecture for every developer in project.

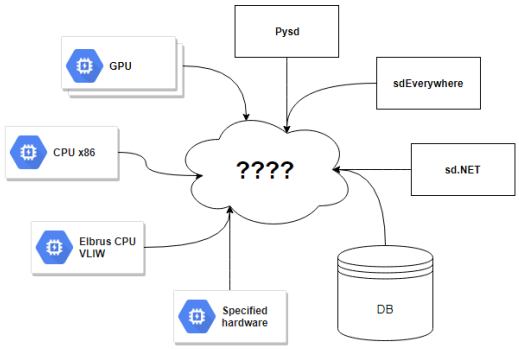

Fig. 2. Existing tools and technologies that already used by sdCloud platform or proposed to use it in future platform development.

3 Concept of a new platform architecture

To solve all described issues and meet requirements presented above, we suggest building solution with micro-services architecture. By this architecture each micro-service communicates with another-ones via Enterprise Service Bus (ESB) – RabbitMQ [9]. Enterprise Service Bus – system that provides a communication between mutually interacting software applications in service-oriented architectures (SOA) [10]. Micro-services architecture is a power tool to build flexible and reliable solutions. Each service can build with own way, with different code languages and different technologies, but only one thing should be a common in each service – communication protocol, and for that role we choose a cross-platform high-performance and reliable ESB system – RabbitMQ [9], [11], [12].

Our first step in architecture designing process is extracting global parts with specified responsibilities. For now our cloud platform provides the following features and functions:

— Store a model source code;
— Store a compiled versions of user models;
— Store model execution results;
— Model execution;

Previously we talk about that cloud platform should be a reliable, so we should also provide some services which should monitor all parts of the system and notify our technical engineers to quick resolve any issues. When we talk about cloud solutions, we assume that all functions can be used by huge number of users. Under users we assume not only peoples, because our system has a public API, users in our system can be external applications and services. For example, it can be an infrastructure under IoT service which implements continuous modeling processes and monitor system. This application generates a lot of input data and load for our computation nodes. So, it was a good to implement special tools to auto-scaling our computation nodes. Those tools can optimize load and implement load balancing between all existing services by optimal way that can reduce delay time between when user starts a model and when results are provided to a user; also, it can reduce power usage of data center where our platform was hosted. Summarize what was said, we can reveal following common parts:

— Input-Output (IO) zone;
— Tool zone;
— Controller zone;
— Monitor zone.

Those common parts of our solution grouped based on responsibilities of services, so we can say that it is a responsibility zones, or just zones. Each of presented zones can contains not only one micro-service, it can contain many different services, but each service should implement only one function that fits in determined responsibility zone.

— IO – zone of services that works with data storages, like a model results persistence storage, model source files storage;
— Tools – zone of services which implements a specific feature-tool to work with models or model results data, like a PySD to produce model results based on user input and model source file;
— Controller – zone of services that control all requests to process data, organize load-balance, implements data stream routing, security checks and some other important staff;
— Monitor – zone of services that checks statuses of all connected to cloud services and communicates with controller services to reorganize cloud structure if something wrong.

The pros of such approach, that it gives to us possibility to deploy services on many computation nodes of different types and organize communication

between them via ESB, based on standard TCP/IP network stack. It allows to us manage each zone separately and implement different algorithms for load balancing and recovery based on different data for each zone in order to use optimal technics for those purposes. For example, for controller zone we can use round-robin algorithm for load-balancing each inner service, because most of them tasks which implemented in those services are simple, so this algorithm is best for this kind of tasks. However, in tools services we have many kinds of tasks where execution time of them depends on various input data, like model complexity, type of tools etc. so we should use different way for load balancing input jobs from users. With new architecture it can be easy to be done, just introduce a new load-balancing service in controller zone to manage jobs for tools. Implementation details of algorithm for load balancing user jobs for system dynamics tools are beyond the scope of this paper.

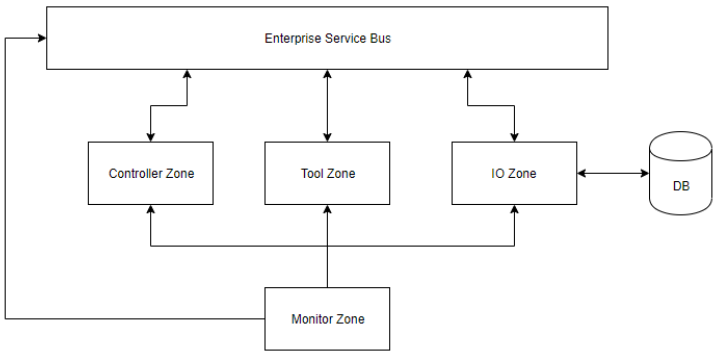

Fig. 3. The scheme of relationships between service zones and Enterprise Service Bus.

4 Communication protocol for Enterprise Service Bus

Let's talk about communication process between zones via ESB. Each service in specified zone should be connected to ESB and should can post new messages and receive messages from them that can be handled by this service. Because most of our processes are time consuming, so we can't use synchronous request-response scheme of communication. For example, model execution process can take in average 5 minutes for medium complexity models and configurations, and for huge complexity models it can take time in 1 hour or more... However, at the same time we should have opportunity to handle synchronous requests,

like taking a status of execution process. For clarifying this process, we should reflect this kind of communications on protocol level. For that purposes we choose a CQRS (command-query responsibilities segregation) or CQS (command-query separation) principle [13]. This principle introduces three basic terms in communication protocol:

1. Command – asynchronous action, that should be performed by target service, and source service don't know when this action was completed. Commands can't return any results data, but can generate new commands, queries and events;
2. Query – synchronous method, that should be executed immediately and provides results data as soon as possible. Queries can't generate new commands or events but can use other queries to take an additional information from another services;
3. Event – immutable data item, that generated by commands to notify services about command progress and providing an additional information about that.

The next key question of communication process – how messages should be routed? The RabbitMQ implements a simple, but very powerful message management. We can define queues and exchanges. Services can post messages in queues and exchanges and in the same time can listen only queues for new messages. Most important thing for ESB – bindings for exchanges. This feature provides messages routing from queues to specified exchange or from exchange to any queue based on route keys. This approach provides to us flexible way for message management. Another key feature of RabbitMQ – manage all ESB infrastructure at real time. We can create any exchanges, queues and bindings on the fly when RabbitMQ cluster already ran and already has many consumers. We can use this feature for manage our cloud platform environment.

In our architecture we plan to use an auto-configurable system. Each microservice in our architecture can have few input queues and few output exchanges. Information about that should be written in specified scheme file and provided for routing-controller service. This scheme provided to this service via ESB. These scheme file also contains information about type of commands and events that service can handle.

Our architecture intends few global queues for providing a generic bus for providing public messages like events and commands. It allows to use a common way to publish new events and commands and configure a correct routing of messages to required services.

For example, looks on simple schema files for two simple services: the first – model executor, implemented in tool zone; and the second one – time frame persistent service, implemented in IO zone.

```
{ 'service': 'sdcloud.tool.xmileexecutor',
  'type': 'tool',
  'input': [
    'sdcloud:model:xmile',
    'sdcloud:model:langc'
  ],
  'output': [
    'sdcloud:timeframe',
    'sdcloud:model:langc'
  ],
  'commands': [
    'command:sdcloud-execute-xmile',
    'command:sdcloud-execution-stop'
  ] }
{ 'service': 'sdcloud.io.timeframe-saver',
  'type': 'io:output',
  'input': [
    'sdcloud:timeframe'
  ]}
```

This schema file provides information about service, which commands and events can be handled by this service and provides information about which data can be consumed and generated by this service. For example, the executor service consumes source code of system dynamics model files or already transpiled versions of them into C language code. Also, this service provides an output data represented as model results time frames. Based on this information we can manage our services and route all required information between services in easiest way.

The next key question is how to organize synchronous requests for data retrieving? We have two ways to do that. One of them – use HTTP based API in required services. By this approach we can use already implemented protocol for Request-Response communication between services. Also, we can use various libraries to make our requests between services in easy way. However, we should implement a request routing between services in another way. This way is not flexible and non-generic, it introduces new technology for communication and ignore whole ESB infrastructure that already solve all communication issues (message routing). So, let's look on the second way how we can implement a Request-Response communication process via ESB. Require introducing a new

special message types which should contain information about sender and message identifier, then just place it in special queue. For that approach we should introduce a new queue for each service – "responses", a new global exchange "requests" which implements a simple routing rules between services. We should introduce two types of route-key identifiers for implementing a best routing:

— Service Pool Identifier – represents an identifier for pool of service instances, f.ex.: "sdcloud:tool:xmileexecutor";
— Service Instance Identifier – represents an identifier of specific instance of service, f.ex.: "sdcloud:tool:xmileexecutor:af8fe3462".

When request message will be sending by pool identifier, then message route to least loaded service instance. The response message will contain identifier of responded service instance, so sender service can continue communication process with that service instance or send other messages again with pool identifier.

To organize correct identifying of request-response pair require to introduce a unique message identifier. The simplest way to done that – use GUID (Global Unique Identifier). This identifier should be generated on sender side.

Sum up, we can reach our goals using ESB: flexibility – we can introduce new services with different technology stacks, only with one common part which is ESB communication protocol; reliability – ESB implementation by RabbitMQ is high reliable solution, which supports clustering and message persisting; scalability – we can run our services with many copies and ESB route all tasks between them with build-in load-balancing algorithm.

5 Responsibility zones

Responsibility zones means a group of micro services that responds only for one global feature, like IO or tool. Every responsibility zone can be managed separately.

5.1 Input-Output zone

Input-Output (IO) responsibility zone is a group of micro services which implements features related with data persisting. In our platform we need to store various types of data that generated by tools and users. Basically, we have two key data kinds: files and times frames. File storage used to store source code of system dynamics models which uploading in our platform by users. Also, we use this storage to store artifacts which generating by our tools. For example, PySD tool generates a transpiled to python version of model source code. Time

frames – part of model execution results data, for optimal store this kind of data we use specialized data base, that calls as Time Series DB. For each kind of data, we implement a separate service

5.2 Tool zone

The following responsibility zone is complex – Tool responsibility zone. This zone implements key features of our cloud platform, for example model execution. This zone has high complexity because integrate many different tools based on different technologies. All those tools form our cloud platform for system dynamic researchers. Because we can host every tool separately, we can deploy services on required platforms that meets the requirements of those tool configuration. For example, for PySD service we should provide platform with already installed python environment. In the feature we can use isolated containers for each tool. When we talk about isolated containers, we assume using of a Docker container. Using containers allows to us easiest way to deploy tools on new platforms and makes a copy of services to improve reliability.

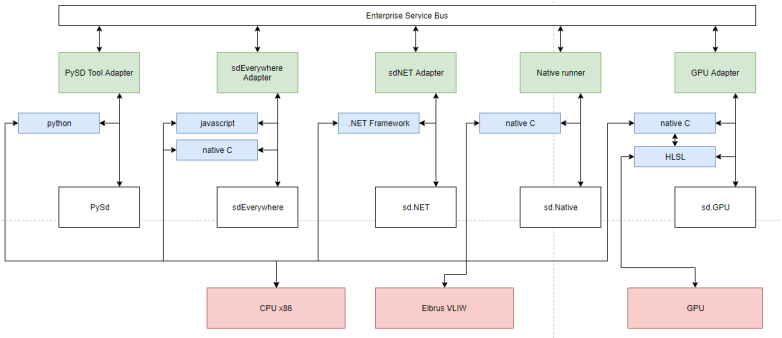

Fig. 4. The scheme of tool services dependencies upon technologies and platform types (green boxes – services, connected to ESB; blue boxes – technologies requirements for tool; red boxes – platform type requirement; white boxes – tools).

With new architecture we can add a new tool in our cloud platform with minimal effort. For that we should make a platform or container with required configurations and implement a specified wrapper. This wrapper should provide communication between ESB and tool via our protocol.

5.3 Controller zone

The controller zone – is a specified zone that should integrate all services from tool and IO zones between themselves. Controller zone composed few common

services. The heart of this zone and whole cloud platform is a Job Routing service. Job Routing service should know information about all ran services in our platform and configure ESB in runtime to organize correct routing of all messages. To harvest information about all new services used another service – Service Discovery service. This service lookup our infrastructure to find new services and receive from them configuration schemas, then this information transferred to Job Routing service via ESB.

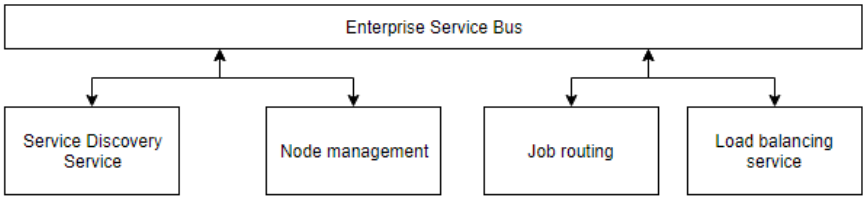

Fig. 5. The scheme of services which used in controller zone.

Additionally, we can add new services in controller zone to improve our infrastructure management, performance, etc. For example, we can implement the Node Management service which have control to startup and shutdown computation nodes if our platform has low load. It can lead to reducing of data center energy consumption.

When we implement the Load Balancing service, we can improve our performance by optimal task scheduling between tool instances. Together with Node Management service we can startup new nodes and rebalance all load between active computation nodes. As a result, waiting time for execution results reduced for our users.

5.4 Monitor zone

The monitor responsibility zone – group of services which should check all services from all zones in our platform. By monitor we assume continuous checks of service status. We should monitor following data for each service:

— CPU usage;
— Memory usage;
— Disk usage;
— Health check response time.

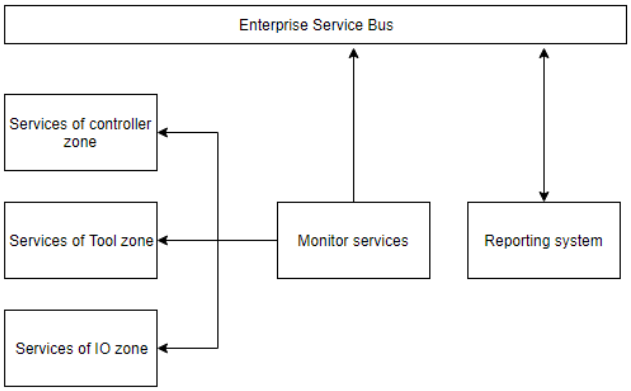

Fig. 6. The scheme of communication process between monitor services and other services in our platform.

Monitor zone structure and communication process looks different in compare with other zones. This relates with different kind of tasks that should be solved by those services. Monitor services should be deployed on each computation nodes separately and provide information about critical events via ESB. The health check request should be implemented via ESB. This way allows to us check how communication are works between monitoring service and ESB. This check should be performed in regular way, for example every 15 seconds. If we found some issues, service should notify technical engineers about it. For that purposes service send specified command to Reporting service which send notification.

6 Conclusion

Nowadays computer modeling area is very popular and mostly researches interest in cloud computing in computer modeling rapidly growing. In scientific world math models with high complexity are continuously developed in different areas of applications, that is a cause of this growth. In this way our team should quick respond for new user requirements and growing of computation complexity for system dynamics models. Previously software architecture of sdCloud platform was complex and hard to maintain and extend. The new architecture based on micro-services infrastructure and Enterprise Service bus provide flexibility, scalability and reliability of computation platform. Previously we don't have a common way to introduce a new tools and integrations that used different technologies. For now, we have a common way for that purposes and this

way provide to us use these technologies in isolated environments. Grouping services by responsibilities makes internal processes transparent and understandable. With new software architecture we reduce time of feature delivery for our users and increase performance and reliability of our platform by starting services with multiple instances.

References

1. Forester W., The Beginning of System Dynamics. USA: MIT, 1989.
2. Zaman G., Jung I. H., "Stability techniques in SIR epidemic models", PAMM: the Proceedings in Applied Mathematics and Mechanics., vol. 7, 2007.
3. Mulyukin A. Adaptation of system dynamics model execution algorithms for cloud-based environment / Mulyukin A., Perl I. // Proceedings of the 22nd Conference of Open Innovations Association FRUCT. - 2018. - Pp. 179-189
4. GitHub website, Source code of PySD by Houghton J., Web: https://github.com/JamesPHoughton/pysd/
5. GitHub website, Source code of SDEverywhere by Fincannon T., Web: https://github.com/ToddFincannon/SDEverywhere
6. Mulyukin, A. Executing system dynamics models on GPU / Mulyukin A., Perl I. // Proceedings of the 36th International Conference of the System Dynamics Society (in the process of publishing)
7. Perl I., Ward R., "sdCloud: Cloud-based computation environment for System Dynamics models", Proceedings of the system dynamics conference, 2016
8. sdCloud: Bringing system dynamics into cloud, website: http://sdcloud.io
9. Shailesh S., Performance analysis of RabbitMQ as a Message Bus / Shailesh S., Jishi K., Purandare K. // International Journal of Innovative Research in Computer and Communication Engineering. – 2018. – Vol. 6 – Issue 1. – Pp. 241-246
10. De Leusse P. Enterprise Service Bus: An overview / De Leusse P., Periorellis P., Watson P. // Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2007
11. Newman S., Building Microservices: Designing Fine-Grained Systems, 1st Edition. O'REILLY, 2015
12. Wolff E., Microservices: Flexible Software Architecture, Addison-Wesley, 2016
13. Debski A. In Search for a Scalable & Reactive Architecture of a Cloud Application: CQRS and Event Sourcing Case Study / Debski A., Szczepanik B., Malawski M., Spahr S. and Muthig D. // IEEE Software – 2016.
14. Mulyukin A., Perl I. "Adaptation of System Dynamics Model Execution Algorithms for Cloud-based Environment", Proceedings of the FRUCT 22 conference, 2018
15. Mulyukin A., Perl I. "Executing system dynamics models on GPU", Proceedings of the system dynamics conference, 2018