Chimera - Simple Language Agnostic Framework for Stand Alone and Distributed Computing

Go Frendi Gunawan
STIKI Malang
Malang, Indonesia
Email: frendi@stiki.ac.id

Mukhlis Amien
STIKI Malang
Malang, Indonesia
Email: amien@stiki.ac.id

Jozua Ferjanus Palandi
STIKI Malang
Malang, Indonesia
Email: jozuafp@stiki.ac.id

Abstract—**Component-Based Software Engineering (CBSE) is a branch of software engineering that emphasizes the separation of concerns with respect to the wide-ranging functionality available throughout a given software system. The main advantage of CBSE is separation of components. A single component only focus on a single task or a related collection of tasks which allowing software developer to reuse the component for other use-cases. By using this approach, software developers can focus on single components rather than dealing with complex monolithic source code. Several approaches have already been developed in order to achieve ideal CBSE. The earliest implementation was UNIX pipe and redirect, while the newer approach including CORBA, XML-RPC, and REST. Unfortunately those approaches are either too complex or lack of features. Therefore, a simple language-agnostic framework, so called Chimera, was developed in this research. Chimera was built on top of Node.js. This framework allows developer to build pipe flow in a chain (a YAML formatted file) as well as defining global variables. Compared to UNIX named and unnamed pipe, this format is easier and more flexible. On the other hand, unlike XML-RPC, REST, and CORBA, Chimera is much simpler. HTTP protocol is only required for distributed computing scenario. Nor it require the components to be aware that they works on top of the framework.**

Keywords*—*Chimera, Language Agnostic, Component-Based Software Engineering, CBSE, Node.js, CLI.

I. Introduction

Component-based software development approach is based on the idea of developing software systems by selecting appropriate off-the shelf components and then to assemble them with a well-defined software architecture [1].

In order to implement Component-Based Software Engineering (CBSE), several approaches have been performed. The earliest attempt was UNIX pipe mechanism [2]. Pipe mechanism was not the only attempt to achieve CBSE. The more modern approaches including XML-RPC [3] and JSON-RPC [4]. Later, Object Management Group (OMG) introduced a new standard named CORBA (Common Object Request Broker Architecture) [5]. Another interesting approach was introduced by Two Sigma Open Source. Two Sigma created a platform known as Beaker Notebook [6] which is mainly used for research purposes. On 2016, Feilhauer and Sobotka introduce another platform called DEF [7].

Despite Unix Pipe, the other mechanisms require the components to be aware that they are parts of the framework. It means that old programs (e.g. *factor* and *calc*) as XML-RPC or CORBA component. At least, additional layer and adjustment have to be built.

Moreover, CORBA, XML-RPC, SOAP, and JSON-RPC also need HTTP protocol since they were designed for client-server architecture. It imply that the developers need to build a web server for being able to use the mechanisms. However, in any use-case that only need a single computer, this is not ideal.

The goal of this research is to provide simple language agnostic framework that allows developer to:

- Reuse old components without building any additional layer.
- Define process flow in readable and straight-forward format.
- Write the process flow in few lines of code.
- Work on both, stand alone and distributed environment seamlessly.

The framework was named Chimera. Compared to CORBA, DEF, XML-RPC, and JSON-RPC, Chimera is much simpler since HTTP is required for distributed computation only. Chimera also use CLI mechanism that works in almost all OS and in most programming language. The only dependency of Chimera is Node.js and several NPM packages.

II. Previous Research

This section presents an in-depth discussion on the previous CBSE implementation, preceeding Chimera.

A. UNIX Pipe

The very first implementation of CBSE was UNIX pipe mechanism [2] which allowed engineer to pass output of a single program as an input for another program. Since a lot of server was UNIX or linux based, this pipe mechanism availability was very high. Even, DOS also provide similar mechanism [8].

Pipe mechanism work by allowing a program's standard output being used as another program's standard input. By placing several programs into a single pipeline, a more complex process might be composed as depicted in figure 1.

For example, consider two different programs, *factor* and *calc*. When a single argument is given, *factor* presents all

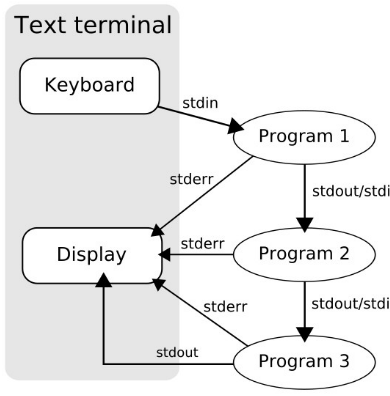

Fig. 1. Unix Pipeline Mechanism

factors of a number. Next, when a single argument is given, *calc* presents the result of an arithmetic operation. Listing 1 depicts the output of both commands.

Listing 1. Usage of factor and calc
```
#! factor 20
20: 2 2 5

#! calc 7+3
10
```

Unix pipe mechanism allows you to combine those two programs. For example, if you want the output of *calc* become the input of *factor*, you can use pipe command as shown in listing 2

Listing 2. Unnamed pipe example
```
#! calc 7+3 | factor
10: 2 5
```

Beside of its high availability and simplicity, UNIX pipe also supports parallel processing through named-pipe mechanism. The named-pipe mechanism can be used to provide cheap parallel processing [9]. As presented in listing 3 a named pipe called *backpipe* was made by using *mkfifo* command. Then, the standard output of *calc* and *factor* is added into the *backpipe*. Finally, the content of the *backpipe* is presented using *cat* command.

Listing 3. Named pipe example
```
#! mkfifo backpipe

#! calc 7+3 > backpipe | factor 20 > backpipe
    | cat backpipe
10
2 2 5
```

Although pipe mechanism provides high availability and capability, it has several limitations. For example, named-pipe mechanism needs external file, which has to be deleted once the operation performed, as a temporary container. This approach does not work straight forward, thus, some efforts needed in order to build a working named-pipe-based-computation.

Pipe mechanism is suitable for simple use-cases involving a single computer. However, at some point, memory sharing and

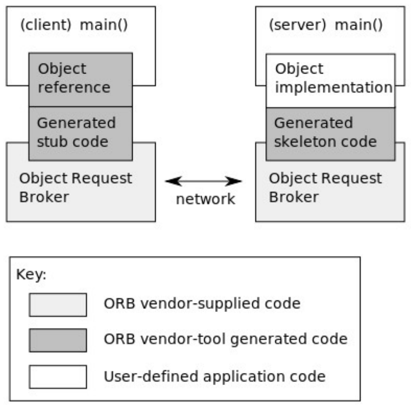

Fig. 2. Object Request Broker

network access are needed when the program become more complicated. Using a mere pipe mechanism to support those requirement needs a lot of efforts. Although it is possible, the readability of the script is severely reduced.

B. CORBA

From CORBA official website, CORBA is defined as standard created by the Object Management Group designed to facilitate the communication of systems that are deployed on diverse platforms [5]. CORBA 1.0 was released on August 1991. The last version, CORBA 3.3, was released on November 2012 [10]. CORBA is heavily affected by object oriented paradigm.

The main component of CORBA is the Object Request Broker (ORB) which acts as a bridge between client and service provider. The service provider (server) provide an implementation of an object, while the client can be a user interface that depend on the service provided by the server. Both, client and server need to agree about the object structure. This agreement is written in an Interface Description Language (IDL), the IDL which is called skeleton in server side, and is called stub in client side.

IDL can be written in Java, C++, or any other language, depending on the implementation of the ORB. An IDL example is shown in listing 4. The interaction between ORB, server, and client depicts in figure 2.

Listing 4. CORBA IDL Example in C++
```
module Finance {
  typedef sequence<string> StringSeq;
  struct AccountDetails {
    string      name;
    StringSeq   address;
    long        account_number;
    double      current_balance;
  };
  exception insufficientFunds { };
  interface Account {
    void deposit(in double amount);
    void withdraw(in double amount) raises(
        insufficientFunds);
    readonly attribute AccountDetails details;
  };
};
```

Compared to UNIX Pipe, CORBA has more complex and rich features. Unfortunately, this also means that CORBA is also more complex than UNIX Pipe. The developer needs to embrace OOP paradigm as well as being familiar with IDL and the CORBA architecture. Despite of it's language agnoticism, some non OOP language (e.g: Matlab and GNU Octave) is not supported by CORBA [7]. CORBA also suffers of several criticism [11]. Regardless to it's popularity, there are many critics towards OOP, as the foundation of CORBA [12].

C. XML-RPC, SOAP, and JSON-RPC

XML-RPC is a specification and a set of implementations that allows software running on disparate operating systems and in different environments to make procedure calls over the Internet. XML-RPC use HTTP as the transport and XML as the encoding. It is designed to be as simple as possible, meanwhile it allows complex data structures to be transmitted, processed and returned [3]. SOAP stands for Simple Object Access Protocol. SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment [13]. SOAP was built on top of XML-RPC. It uses XML format as well as HTTP protocol.

JSON-RPC is lightweight remote procedure call protocol similar to XML-RPC [4]. The main difference between XML-RPC and JSON-RPC is the data transfer format. In most cases, JSON is more lightweight compared to XML.

XML-RPC, SOAP, and JSON-RPC heavily depend on HTTP for their inter-process-communication protocol. This is ideal for client-server architecture as HTTP is quite common and easy to be implemented. Those three methods are basically another implementation of RPC (Remote Procedure Call). Compared to CORBA, these three methods are more flexible. With the exception of SOAP, they don't enforce developer to embrace OOP paradigm.

In terms of language agnoticism, XML-RPC and JSON-RPC support any languages that can access HTTP and parse/create the data format. However, in order to use these protocols, a developer should be aware that the components they built work as a part of a bigger system. Tools or programs that built without this consideration need some adjustments or additional layers in order to make them work with the protocol. For example, using *factor* or *calc* as components of XML-RPC might require developer to build another program to catch the output and wrap it in XML envelope.

D. DEF

DEF is a programming language agnostic framework and execution environment for the parallel execution of library routines [7]. DEF focuses on parallel processing by enabling shared memory and message passing. DEF needs several components and so JSON as data exchange format. DEF is better in terms of parallelism and language agnosticism compared to CORBA, Matlab, and Parallel Fortran. CORBA for example, doesn't support matlab and octave [7].

However, DEF still depends on HTTP for inter-process communication. Consequently, in order to build DEF architecture, a web server is needed. Also, the developer needs to make sure that each components aware of the architecture. As in CORBA, XML-RPC, SOAP, and JSON-RPC, additional layer might be needed to make use of old components.

E. Beaker Notebook

Beaker Notebook [6] is also considered as an interesting approach of CBSE. The platform was developed by Two Sigma Open Source for research purposes only. Beaker Notebook provides native autotranslation that lets a developer declare specific variables in a cell in one language, and access them seamlessly in a different cell and language.

Llisting 5 describes a 6 by 4 table populated with random numbers. The table is, then, saved as global variable df. The data is loaded and showed in listing 6.

Listing 5. Beaker Python Cell Example
```
import pandas
beaker.df = pandas.DataFrame(np.random.randn
    (6, 4), columns = list('ABCD'))
```

Listing 6. Beaker R Cell Example
```
beaker::get('df')
```

Moreover, Beaker Notebook is good for prototyping. It also has a very simple API compared to CORBA or XML-RPC. However, it still require the developer to add additional layer in order to use old components like *factor* or *calc*

III. Chimera Goal

Our goal was to build framework that has excellent backward component compatibility, format readability, as well as distributed and stand alone compatibility.

A. Backward Component Compatibility

Our greatest task was to let developers use the pre-existing components without building too many adjustment layers.

factor, *calc* and most UNIX utilities were excellent example for this case. Those programs were commonly used before CORBA, XML-RPC and JSON-RPC were built.

CORBA, XML-RPC, and JSON-RPC cannot directly use those old components. Additional layer has to be built. The developer has to reformat the output of the components into XML or JSON object.

B. Readability of The Format

C. Small Line of Codes

D. Distributed and Stand Alone Interoperability

- Reuse old components without building any additional layer.
- Define process flow in readable and straight-forward format.
- Write the process flow in few lines of code.
- Work on both, stand alone and distributed environment seamlessly.

IV. Chimera Architecture

From the previous section, it can be concluded that Unix Pipe Mechanism was the simplest one despite of it's lack of features. We also notice that Beaker Notebook's like memory-sharing mechanism is much simpler compared to CORBA and other network-based protocols.

The objective of this research is to make a very simple framework that is truly language agnostic. The framework should be able to accomodate with old components and should not enforce developers to embrace any particular programming paradigms. Moreover, making unnecessary new standard is avoided in this research. By making use of technologies that most developers are familiar with, it is expected that the adaptation is going to be easier. It is assumed that most programming languages supports command line interface and command line arguments. By creating a framework that depends on command line protocol, maximum language agnosticism with less effort is achieved.

In figure 3 we show the architecture of Chimera. Suppose *Program1* and *Program2* should run in parallel, and *Program3* should be executed once *Program1* and *Program2* finished.

Chimera architecture consists of several parts. The Chimera Core is the main component that responsible for orchestrating external programs into a single process flow. Chimera Core can access temporary memory which is no other than a JSON Object placed in memory. All input and output of external programs are copied into this JSON Object. The process flow itself written in a YAML Chain File. YAML is a common format for configuration. YAML depends on indentation and whitespace, making the format easily readable. Using these three components, Chimera is able to do everything that UNIX Pipe is able to do. In fact, developers can even define the UNIX Pipe inside Chimera. The other two components are Chimera-Service and Chimera-Sender. These two components are responsible for HTTP communication. Chimera-Service and Chimera-Sender brings the entire framework into distributed environment.

Figure 3 describes the process started when the user ask Chimera to execute the process. User should provides the YAML file location and the process inputs. After getting a request from user, Chimera read the YAML file, retrieving content, and initiating global variables in temporary memory. The framework, then, executes external programs sequentially or parallel, depending on the content of the YAML file. When the external programs are located in the same computer, Chimera-Core executes the programs and provides the required stdin parameters which are taken from temporary memory. After the program executed successfully, Chimera-Core reads the stdout and saves it in the temporary memory for further process.

In case the programs are located in different computer, the user should provide invocation of Chimera-Sender then Chimera-Sender contacts Chimera-Service to run Chimera-Core remotely. Once the process completed, Chimera-Service sends response back to Chimera-Sender. At last, Chimera-Sender sends the response back to the local Chimera-Core.

At the end of the process, Chimera returns the output of those chain-processes to the user.

V. Chimera Technical Implementation

Chimera as an NPM package has been published and accessible at https://www.npmjs.com/package/chimera-framework.

A. Node.js and NPM As Chimera's Foundation

Chimera was written in Node.js. Node.js itself is a JavaScript runtime built on Chrome V8 JavaScript engine. Node.js uses an event-driven, a non-blocking I/O model that makes it lightweight and efficient [14]. Compared to Python and PHP, Node.js has an overall better performance [15]. Node.js is also available for Windows, Linux, and Mac.

Node.js has a package-manager named NPM (Node Package Manager). This allows developers to use libraries that have already been written by other developers. Chimera depends on several packages:

- async
- express
- fs-extra
- http
- js-yaml
- node-cmd
- path
- process
- querystring

Async package was used to develop the control flow since Node.js has non-blocking mechanism. Express which allows us to catch request and to send appropriate response was used to build Chimera-Service. Fs-extra was also used in order to do file operations. Http and querystring were used to build Chimera-Sender. Several functionalities provided by node-cmd were needed in order to execute other programs through CLI mechanism. Js-yaml was used for YAML parsing. At last, path and process were needed to determine absolute path of a file as well as to change directory and retrieve input arguments.

B. YAML for Defining Chain

YAML (YAML Ain't Markup Language) standard, when was built in 2001, is a human friendly data serialization standard for all programming languages [16]. Unlike JSON, YAML depends heavily on indentation. Although its size is bigger compared to JSON, YAML is readable and commonly used for configuration.

At first, Chimera's chain file was intended to use JSON format. However, the use of JSON format might not be a good decision since the developers have to be very careful with commas and curly braces. Another disadvantage of JSON is that it doesn't provide any intuitive way to add comments, which is quite essential in writing algorithms.

Let consider having several programs written in Python, Java, PHP, and Javascript. Each of them takes two arguments, does simple arithmetic operation, and return a single output. Given a and b, to be calculated ((a+b) * (a-b)) + a.

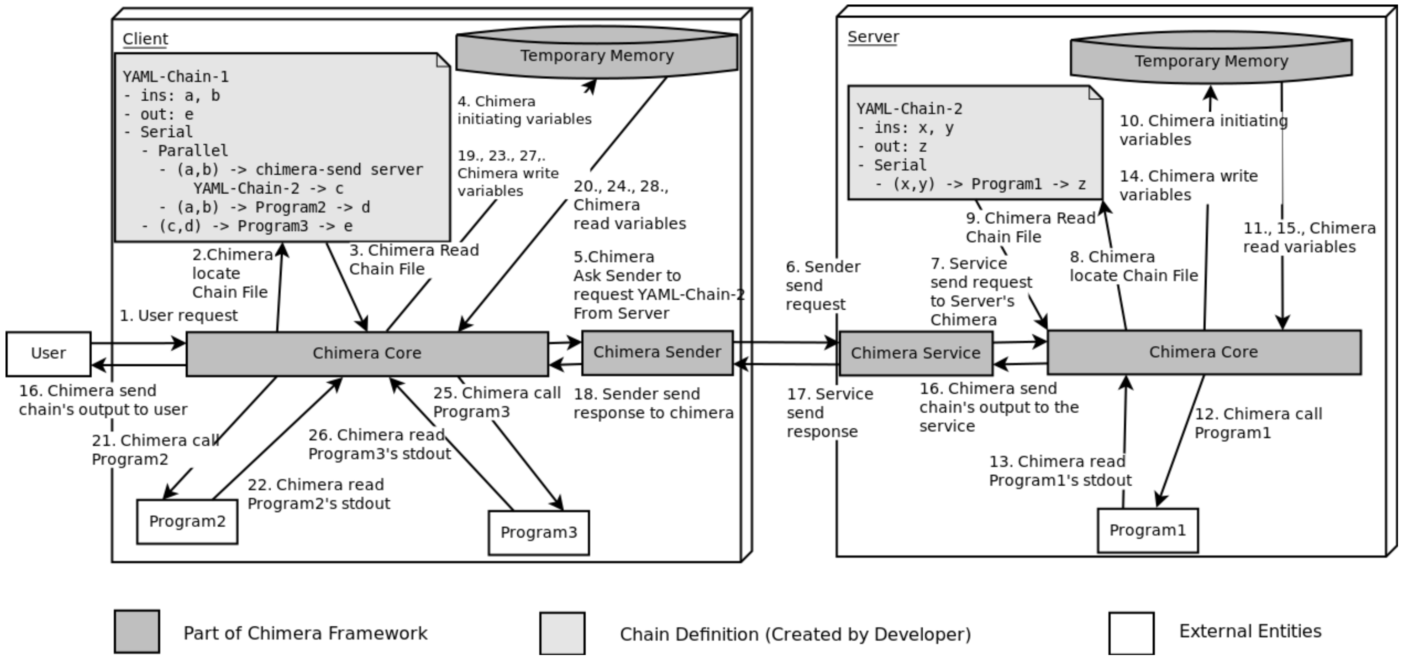
Client
YAML-Chain-1
- ins: a, b
- out: e
- Serial
  - Parallel
    - (a,b) -> chimera-send server
        YAML-Chain-2 -> c
    - (a,b) -> Program2 -> d
  - (c,d) -> Program3 -> e
Temporary Memory
4. Chimera initiating variables
19., 23., 27,. Chimera write variables
20., 24., 28., Chimera read variables
2.Chimera locate Chain File
3. Chimera Read Chain File
5.Chimera Ask Sender to request YAML-Chain-2 From Server
1. User request
User
Chimera Core
Chimera Sender
16. Chimera send chain's output to user
25. Chimera call Program3
18. Sender send response to chimera
21. Chimera call Program2
26. Chimera read Program3's stdout
22. Chimera read Program2's stdout
Program2
Program3
Server
Temporary Memory
YAML-Chain-2
- ins: x, y
- out: z
- Serial
  - (x,y) -> Program1 -> z
10. Chimera initiating variables
14. Chimera write variables
9. Chimera Read Chain File
11., 15., Chimera read variables
6. Sender send request
7. Service send request to Server's Chimera
8. Chimera locate Chain File
Chimera Service
Chimera Core
17. Service send response
16. Chimera send chain's output to the service
12. Chimera call Program1
13. Chimera read Program1's stdout
Program1
Part of Chimera Framework
Chain Definition (Created by Developer)
External Entities

Fig. 3. Architecture of Chimera

The process csan be written as follow:

*f = ((a+b) * (a-b)) + a*

Then, this process is divided into several sub-processes:

- Process 1: c = a + b
- Process 2: d = a - b
- Process 3: e = c * d
- Process 4: f = e + a

Process 1 and process 2 are executed parallelly since they are independent to each other. It is not necessary to solve process 1 first in order to do process 2 and vice versa.

After process 1 and process 2 finished, process 3 and process 4 should be executed sequentially. Process 3 depends on process 1 and 2, while process 4 depends on process 3

Listing 7 is the example of YAML formatted chain file.

Listing 7. YAML Chain Example
```
ins: a,b # The inputs of main process
out: f # The outputs of main process
series:
  # Process 1 and 2
  - parallel:
      # Process 1 (in Python)
      - ins: a, b
        out: c
        command: python programs/add.py
      - series: # Process 2 (in Java)
          # First, compile the source
          - javac programs/Substract.java
          # then run the program
          - ins: a, b
            out: d
            command: java -cp programs
                Substract
  # Process 3 (in PHP)
  - ins: c, d
    out: e
    command: php programs/multiply.php
  # Process 4 (in Javascript)
  - ins: e, a
    out: f
    command: node programs/add.js
```

Similar to other programs, semantically, each node in YAML contains *ins* and *out* which means that the program might have more than one inputs but it only provide one single output. The last element of the node is either *command*, *parallel*, or *series*. If the process has no other sub-process, *command* is used. Otherwise, if the process contains another sub-process, *parallel* or *series* is used depending on the desired execution flow. The best practice is that parallel flow is recommended, when two or more processes are independent on each other. Listing 8 provides single-line shorthand for *ins*, *out*, and *command*. *ins* should be written inside parantheses, while -¿ act as separator between *ins*, *command*, and *output*. Listing 8 is equal to Listing 7.

Listing 8. YAML Chain With Shorthand
```
ins: a,b # The inputs of main process
out: f # The outputs of main process
series:
  # Process 1 and 2
  - parallel:
      # Process 1 (in Python)
      - (a,b) -> python programs/add.py -> c
      - series: # Process 2 (in Java)
          # First, compile the source
          - javac programs/Substract.java
          # then run the program
          - (a,b) -> java -cp programs
              Substract -> d
  # Process 3 (in PHP)
  - (c,d) -> php programs/multiply.php -> e
  # Process 4 (in Javascript)
  - (e,a) -> node programs/add.js -> f
```

C. JSON Format for Temporary Global Storage and Data Transfer

When Node.js is used, it is common to use Javascript Object Notation as Chimera's global storage and network data transfer. JSON (JavaScript Object Notation) is a lightweight data-interchange format [17]. Once the YAML Chain File parsed, Chimera creates several variables in *name : value* pairs. For example, after loading YAML in Listing 7, the global variables contain JSON object as shown in Listing 9. As the sub-process executed, several others variables will be added as needed. At the end of the process, the global storage contains a, b, c, d, e, and f.

Listing 9. Initial content of JSON Storage
```
{
    "a" : 0,
    "b" : 0,
    "f" : 0,
}
```

Not only for temporary storage, we also use JSON for data transfer between Chimera-Sender and Chimera-Service. The data sent to Chimera-Service is shown in Listing 10, while the data received by Chimera-Sender is shown in Listing 11.

The JSON request contains two keys, i.e. the "chain" for indicating the remote YAML chain file location and "input" containing array of inputs.

Meanwhile, the JSON response contains three keys, i.e. the *success* key for indicating whether the request succeed or failed, *errorMessage* containing the error message and *response* is the response from the server.

Listing 10. JSON Request
```
{
    "chain" : "remote-chain-file.yaml",
    "input" : [],
}
```

Listing 11. JSON Response
```
{
    "success" : true,
    "errorMessage" : "",
    "response" : "",
}
```

D. Utilities

Several utilities were built as the components of Chimera Framework, such as:

- Chimera-core
- Chimera-eisn
- Chimera-serve
- Chimera-send

Chimera-core is the main component of the framework in which user can invoke Chimera by executing *chimera your-chain-file.yaml [input1 [input2] ...]*. In terms of convenience, some shorthands are also provided, e.g. the use of chimera *chimera "command:cal"* or even *chimera "cal"* to call Chimera.

Chimera-eisn takes at least three input arguments. The first and the second argument should be file name, while the third arguments should be the command. The command is executed only when the first argument's modification date is newer than the second argument. EISN itself stands for "Execute If Source Newer". This is useful when a user wants to use source code of compiled language as an argument as described in Listing 12

Listing 12. Chimera-eisn usage example
```
ins: a, b
out: c
series:
    - chimera-eisn add.java add.class javac
        add
    - ins: a, b
      out: c
      command: java add
```

Chimera-serve is a utility to allow several chain files being served by a computer. The typical usage of chimera-serve is:

TIMEOUT=5000 PUBLISHED=. chimera-serve

The first two statements are used to define timeout and to publish directory. Dot means current directory. All chain files in published directory are accessible over the network.

Chimera-send is a utility to access chimera service. The typical usage of chimera-send is:

TIMEOUT=5000 chimera-send remoute-chain-file.yaml [input1 [input2] ...]

VI. Test

Listing 13 describes the process of testing whether the Chimera works in parallel and distributed environment.

Listing 13. Distributed and Parallel YAML-chain Scenario
```
ins: a,b,server
out: e
verbose: true
series:
  - parallel:
      # P 1
      - (a,b) -> php programs/add.php -> c
      # P 2
      - (server, 'tests/chain-add.yaml', a,b)
          -> chimera-send -> d
  # P 3
  - (c,d) -> node programs/add.js -> e
```

The chain consisted of three processes. Process 1 and process 2 was executed in parallel. After process 1 and process 2 finished, process 3 was executed. Unlike process 1 and process 3, process 2 will ran on the server.

On the server side, *chimera-serve* was executed so that it was able to listen the client's request and to give response as necessary.

Meanwhile, on the client side, it was executed as *chimera tests/chain-distributed.yaml 4 5 http://localhost:3000*. The client log depicts in listing 14

Listing 14. Client Log
```
[INFO] START PROCESS [php programs/add.php
      "4" "5"] AT      : 7,310,467,197,740
[INFO] START PROCESS [chimera-send "http://
     localhost:3010" "tests/chain-add.yaml
     " "4" "5"] AT      : 7,310,489,588,596
[INFO] END PROCESS   [php programs/add.php
      "4" "5"] AT      : 7,310,529,490,373
[INFO] PROCESS       [php programs/add.php
      "4" "5"] TAKES : 62,228,930 NS
[INFO] STATE AFTER   [php programs/add.php
      "4" "5"]         : {"a":4,"b":5,"server":"
     http://localhost:3010","c":9}
[INFO] END PROCESS   [chimera-send "http://
     localhost:3010" "tests/chain-add.yaml
     " "4" "5"] AT      : 7,310,786,490,287
[INFO] PROCESS       [chimera-send "http://
     localhost:3010" "tests/chain-add.yaml
     " "4" "5"] TAKES : 296,877,900 NS
[INFO] STATE AFTER   [chimera-send "http://
     localhost:3010" "tests/chain-add.yaml
     " "4" "5"]         : {"a":4,"b":5,"server":"
     http://localhost:3010","c":9,"d":9}
[INFO] START PROCESS [node programs/add.js
      "9" "9"] AT      : 7,310,787,707,145
[INFO] END PROCESS   [node programs/add.js
      "9" "9"] AT      : 7,310,930,153,276
[INFO] PROCESS       [node programs/add.js
      "9" "9"] TAKES : 142,419,968 NS
[INFO] STATE AFTER   [node programs/add.js
      "9" "9"]         : {"a":4,"b":5,"server":"
     http://localhost:3010","c":9,"d":9,"e":18}
```

From the log, it can be seen that the second process on client side was executed before the first process finished. However, the third process didn't start before the second process started. The output of process 2 was taken from the server's response.

The benchmark of the framework is shown at Figure 4 and table I.

It can be concluded that in average, process 2 takes the longest time. This is expected, since it was executed in the server.

VII. Conclusion

Chimera, a simple language agnostic CBSE Framework has been developed in this research. The framework is quite simple, yet powerful enough for being used on distributed and parallel computation.

Old UNIX commands, like *calc* and *factor*, work perfectly as well under the framework and no additional layer needed.

Acknowledgment

The authors would like to thank Sonny Setiawan, Satriyo Wibowo, Dani Devito, and Zusana Pudyastuti for their suggestions and inputs.

TABLE I. Chimera Framework Benchmark

Experiment	Process	Start (ms)	End (ms)	Time (ms)	Total Time (ms)
	P 1	0	62.29	62.29	
1	P 2	22.39	319.29	296.9	501.64
	P 3	320.51	462.96	142.45	
	P 1	0	62.22	62.22	
2	P 2	18.97	260.11	241.14	447.92
	P 3	261.35	405.9	144.55	
	P 1	0	56.02	56.02	
3	P 2	20.29	238.06	217.77	415.51
	P 3	239.62	381.33	141.71	
	P 1	0	55.54	55.54	
4	P 2	18.48	244.95	226.47	426.7
	P 3	246.14	390.84	144.7	
	P 1	0	55.39	55.39	
5	P 2	26.02	252.57	226.55	431.5
	P 3	253.78	403.35	149.57	
	P 1	0	53.84	53.84	
6	P 2	18.92	254.81	235.89	430.65
	P 3	256.07	396.99	140.91	
	P 1	0	55.14	55.14	
7	P 2	20.39	249.6	229.21	430.32
	P 3	250.74	396.7	145.96	
	P 1	0	50.33	50.33	
8	P 2	20.15	231.62	211.47	403.53
	P 3	232.79	374.52	141.73	
	P 1	0	54.57	54.57	
9	P 2	18.88	238.59	219.71	418.71
	P 3	240.2	384.62	144.43	
	P 1	0	55.21	55.21	
10	P 2	18.87	246.38	227.52	423.98
	P 3	247.72	388.97	141.26	
Total Execution Time Average					433.046
P1 Execution Time Average					56.057
P2 Execution Time Average					233.263
P3 Execution Time Average					143.276

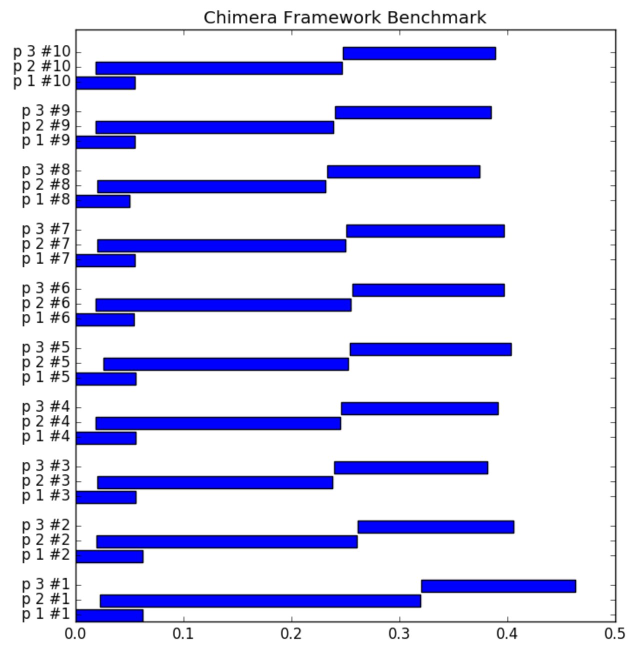

Fig. 4. Chimera Framework Benchmark

References

[1] A. Kaur and K. S. Mann, "Component based software engineering," *International Journal of Computer Applications*, vol. 2, no. 1, pp. 105–108, 2010.

[2] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell, "Mass-produced software components," in *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, 1968, pp. 88–98.

[3] Trac, "Json-rpc," http://json-rpc.org, accessed: 2017-05-30.

[4] I. UserLand Software, "Xml-rpc.com," http://xmlrpc.scripting.com, accessed: 2017-05-30.

[5] O. M. Group, "Corba," http://www.corba.org/, accessed: 2017-05-30.

[6] L. T. Two Sigma Open Source, "Beaker notebook," http://beakernotebook.com/index, accessed: 2017-05-30.

[7] T. Feilhauer and M. Sobotka, "Def-a programming language agnostic framework and execution environment for the parallel execution of library routines," *Journal of Cloud Computing*, vol. 5, no. 1, p. 20, 2016.

[8] B. Watson, "Dos 7 command," http://www.lagmonster.org/docs/DOS7/pipes.html, accessed: 2017-06-19.

[9] T. Conway, "Parallel processing on the cheap: Using unix pipes to run sas programs in parallel. sas users group international (sugi 28) proceedings. march 30–april 2, 2003. seattle, washington," 2003.

[10] O. M. Group, "Corba," http://www.omg.org/spec/CORBA/, accessed: 2017-05-30.

[11] M. Henning, "The rise and fall of corba," *Queue*, vol. 4, no. 5, pp. 28–34, 2006.

[12] I. Hadar, "When intuition and logic clash: The case of the object-oriented paradigm," *Science of Computer Programming*, vol. 78, no. 9, pp. 1407–1426, 2013.

[13] W. X. P. W. Group, "Soap version 1.2," https://www.w3.org/TR/soap12, accessed: 2017-05-30.

[14] N. Foundation, "Node.js," https://nodejs.org/en/, accessed: 2017-05-30.

[15] K. Lei, Y. Ma, and Z. Tan, "Performance comparison and evaluation of web development technologies in php, python, and node. js," in *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on*. IEEE, 2014, pp. 661–668.

[16] C. C. Evans, "The official yaml website," http://yaml.org/, accessed: 2017-05-30.

[17] E. International, "Json," http://json.org/, accessed: 2017-05-30.