3-Way Scripts as a Base Unit for Flexible Scale-Out Code

Marat ZHANIKEEV[1,a)]

Abstract: Distributed and/or parallel code is normally based on elaborate platforms. The main problems with such platforms are (1) constraints placed on operation of the code and (2) overhead imposed by the platform that arbitrates among multiple instances within the running code. This paper argues in favor of platform-less distribution of code. The base unit is referred to as 3-way script, where the three ways are (1) calling a method/function of an instantiated class, (2) executing the code from the command line, and (3) calling a method/function using HTTP requests to a remote web API. The key merit of the proposal is that all the three uses are possible on the same code, which by developer only one – this code is referred to as a 3-way script. This paper discusses examples of the code written in PHP, while the same design is possible in several other popular programming languages.

Keywords: distributed code, 3-way scripting, distributed objects, distributed components, heroku, docker, cloud applications

1. Introduction

This paper meets a recent call for new cloud distribution platforms in [15]. The paper has an excellent review both on traditional and modern platforms, focusing on Ibis and its spinoffs (Constellation, etc.) as an example of a cutting edge platform today. This paper meets the same challenge but with an alternative view point referred to as *distributed code* and implemented as *3-way scripts*.

Traditional tools exists since as far as 1999 with Corba being arguably the first popular tool on the market [7]. There are also Java RMI [13] and OS-native platforms like Cocoa in Mac OS [8], most of which are still used in practice today. The biggest problem with traditional platforms is that they are based on the *client-server* unit of distribution which does not work well in heterogeneous environments like those found in clouds [2]. Traditional platforms also implement networking as continuous sockets which are not feasible under the **scale-out** distribution design popular in clouds today, where there can be hundreds or even thousands of concurrent instances.

Recent advances attempt to resolve these problems by adding *hierarchical structure* [10] or even allowing for P2P networking [13]. Such platforms rely on distributed data storage (normally Distributed Hash Tables (DHT) are used), NAT traversal, and other technologies. In P2P topologies, it is important to create structure on the fly by creating clusters or nodes and assigning superpeers [11].

A separate branch of advanced methods is the Service Oriented Architecture (SOA) [9]. It also relies on distributed storage but is unique in that the distributed network is centered around *tasks* rather than objects. Also known under the name of Service Oriented Computing (SOC), it is still an active area of research [12]. The ultimate destination of SOA/SOC is the Ambient Computing discussed in [14]. Some of the goals of Ambient Computing are discussed further as part of the constraints for the proposed method.

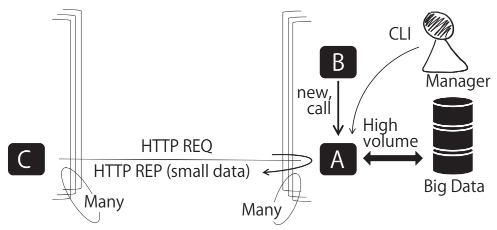

Fig. 1 A common problem in heterogeneous cloud environments resolved by the proposed *3-way scripting*.

Fig.1 shows an example problem from fog computing [2], which explains why there is a need for the new method called **distributed code** in this paper. The example comes from a cloud-based video streaming service where VMs/apps can have fairly large local storage but very low-capacity links to the outside world. In this situation, each node should have at least two behaviors – **local versus remote** in order balance the difference in performance efficiently. The same problem occurs when creating a population of *Docker-based applications* [2], over-the-network interfaces to Big Data processors [3] or a localized intensive processing environment – the example of data streaming on multicore in [4]. Finally, both traditional and advanced platforms do not offer the flexibility for users to generate their own network topologies, where the Virtual Network Embedding (VNE) is a recently introduced technology for optimization of virtual topologies [5].

Let us consider the practical situation in Fig.1 and the functions it requires. Instance A performs a high-volume local job,

[1] Computer Science and Systems Engineering
Kyushu Institute of Technology
Kawazu 680-4, Iizuka-shi, Fukuoka-ken, 820–8502 Japan
[a)] maratishe@gmail.com

potentially at the scale of a locally hosted Big Data. Instance B comes from another code but needs to use the code of A as a library for high-volume local interaction with Big Data. A local human manager and/or software automation also want to use the code of A as a commandline interface. Finally, Instance C comes from the same code/application as Instance A but needs to call A using GET/POST HTTP requests. Note that only the C-A interaction falls under the category of traditional distribution while the remaining functionality is not provided by either traditional or advanced platforms. However, such functionality is crucial for rapid deployment and efficient operation and monitoring of large-scale distributed applications in clouds [2]. The next section introduces the multi-purpose code in Fig.1 as **3-way scripting**.

The above problems have recently been recognized and are found in literature on cloud applications [15]. However, the best offered solution is the Ibis (and its build-up Constellation) platform which is a mixture of traditional, SOA and other components intended for supporting multi-purpose distribution in clouds. Ibis is still task-oriented and therefore belongs to the SOA class but offers more flexibility in general. See the last section for the comparison of specific features with traditional platforms and the proposal.

This paper offers an alternative solution to the same problems that are tackled by Ibis. In fact, the proposed method solves some of the problems marked as *partially resolved* by Ibis in [15]. The core idea in this paper is the notion of **distributed code** as opposed to objects or data. To be efficient, the code has to be minimized and allow for multiple **modes of execution**. Here, the 3-way scripting method – also described in this paper for the first time – offers three separate execution modes and therefore offers a useful technique for implementing the notion of distributed code in practice. The 3-way scripting method is already used in practice as a method for distributing applications across fog clouds [2], but this paper is the first publication that focuses on this particular component as a crucial part in efficient large-scale distribution. The coding advice contained in this paper also comes from practical experience.

2. Distributed Code via 3-Way Scripts

Objectives of the proposed method are the same as formulated for Ambient Computing in [14]. However, this section formulates the objectives focusing on cloud environments and the distributed code concept.

Volatility is formulated in [14] in relation to connectivity, but in clouds it also comes from frequent migrations and changes in populations – the term used to describe a large number of Virtual Machines (VMs) or container-based apps providing the same service (scale-out) [2]. High volatility also demands that distributed code is highly flexible both in handling and in deployment.

Heterogeneous environments in [14] is about wireless spaces while in clouds heterogeneity comes from differences in hardware and network performance across the numerous locations included in a cloud. This problem peaks for fog clouds which, by definition, are located at network edge [2].

Autonomy also refers to wireless spaces in [14] but applies equally well for the practical example discussed in Section 1, where data-intensive local environments can be considered autonomous because they cannot move easily to another location – to be exact, the code can move but the local data and execution environment is unique to a given location and cannot be recreated at another location at a reasonable cost. Note that this side of autonomy is related to heterogeneity as the same code can run in two locations with completely different execution environments and, therefore, practical objectives for each instance.

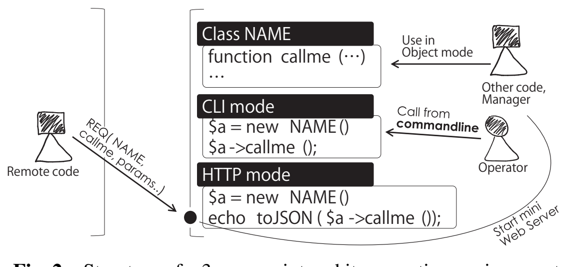

Fig. 2 Structure of a 3-way script and its execution environment.

Fig.2 shows the structure of a single-file 3-way script. The pseudocode is in PHP but the same technique will work in Perl, Python, Ruby and other popular scripting languages. Let us consider the structure in top-down order. The core functionality is written as the traditional *class*, which can be used to create instances. Each *public function* in the class is a unit action. *CLI mode* is a procedural (non-object, outside of the class code) code that uses the *IF...THEN* construct to fork into the CLI mode. The *HTTP mode* is also a fork into the respective mode.

Forking here has the following meaning. In both CLI and HTTP modes, it is assumed that either commandline call or GET/POST request carry the name of the function (action) and parameters necessary for a given action. Each fork is simply an interpreter for these variables. Once interpreted, a new instance of the class is creates and the necessary function is called by reference. *Calling by reference* is easy in PHP and other popular scripting languages – one can use a variable to refer to a function as shown in the following code snippet:

```
$a = new NAME();
$b = 'callme';
$a->$b( $param, $defaultparam = 10);
```

A minor difference between CLI versus HTTP forks is in the format of the output. In CLI mode, the output is in plaintext (*stdout*) in commandline, while HTTP requests would normally require a JS-compatible format like JSON. Current 3-way scripting prototypes use JSON for both as is common with cloud services (Heroku, Docker, etc.), hoping that humans can read the hash array notation. This also makes it possible to parse the standard output by 3-rd party tools.

Using the above design, the 3-way script in Fig.2 can work in the following 3 modes. When used as an *object*, other code (manager, automation, etc.) creates an instance from the class and calls its functions directly. When used in *CLI mode*, a human operator (or, again, software automation) calls a function from commandline – this mode is extremely useful for Docker deployments [2] where initialization from command line is easier to implement (Docker apps are limited to one executable per container).

Finally, when used in *web API mode*, the code needs a web server to receive requests. However, most popular scripting environments (php, python, ruby, etc.) have **mini web servers** built into their execution engines. Starting a web server in such environments is as easy as running a single command without any prior configuration. For sockets (WebSockets for browser clients), a separate server can be written in any scripting language – see example experiments and prototype code for a WebSocket client/server written in PHP at [6].

For *Remote Code* (left side of Fig.2), RESTful requests can be sent using built-in HTTP functions in a given language. However, in cloud applications it is common to use existing tools like *wget* or *curl* and simply parse the JSON in replies.

Advantages of the proposed method over traditional and advanced (Ibis, etc.) platforms are as follows. The code is a standalone script and needs no platforms other than the engine for the scripting language itself. The single-file feature is important for clouds where most services are relatively small (scale out can be rephrased as *small code, many copies*) and easily fit into a single file. Since the script is *both autonomous and multi-purpose*, it can be part of a wide array of possible topologies. The proposed method is part of a management framework in which topology is optimized dynamically using the Virtual Network Embedding (VNE) approach [5] – specifically the cited paper considers a networks built from *hub-and-spokes* basic units. The proposed method is also part of small-scale but extremely performance-intensive environments in [3] and [4].

3. Web API Security

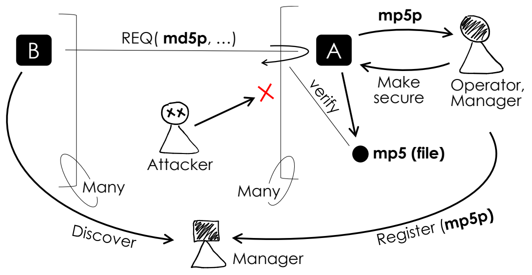

Fig. 3 Security in distributed networks of 3-way scripts.

When running web (RESTful) APIs in a large number of nodes in the Internet, security is a top priority. Fig.3 shows a very simple yet a robust security design. Following the stated earlier objectives, the design does not restrict flexibility of the 3-way scripts.

Step 1 is when local manager (or deployment automation) tells the code to secure itself. The code uses some unique information about itself plus a random component to generate an MD5 hash. The hash is stored in local filesystem while its prefix (*md5p*) is returned to the calling party. Prefix can be of any length up to the full length of the hash, however, it is advised to limit the length to 10-15 symbols for manual operators. Note that multiple keys can be created for different users to enable logging and later performance/fault analysis for each remote node.

Step 2 is when the *md5p* prefix is registered with an external manager (or cluster head in hierarchies as in [11]) where it can be *discovered* by other nodes of the manager itself. The security is enforced simply by ignoring all API calls which come without or with a wrong key. The solution is flexible because the code can be easily toggled between secure and insecure states – *togglesecure()*, for example, can be used as a toggle function for code security.

Note that this technique is not a solution against DDoS attacks. However, randomizing or dynamically changing listening ports can partially resolve the DDoS problem as well. Resilience to DDoS attacks is not an immediate objective of this paper.

4. Comparison and Discussion

Table 1 Feature comparison between the proposed concept of *distributed code* and existing platforms.

	(Web) Sockets	RESTful calls	Hierarchical structure	Heterogeneous environments	Learning curve (simplicity)	Needs a platform?	Traffic encoding optional?
Traditional (Corba, ESB, MPI,...)	YES	NO	NO	NO	NO	YES	NO
Advanced/modern (SOC, Ibis, ...)	YES	YES	YES	YES	NO	YES	NO
Distributed code (proposed)	**YES**	**YES**	**YES**	**YES**	**YES**	**NO**	**YES**

Table 1 shows the comparison between *traditional* (Corba, ESB, RMI, etc. as per discussion in [7][8]), *advanced* (SOA, Ibis, etc. as per discussion in [15]), and the proposed method with the 3-way scripting implementation as the representative of the proposed method in practice. This section offers comparison at two levels, first discussing features which are shared with the advanced methods and then proceeding to the features unique to the proposed method. The table left-to-right advances in the same order, listing the unique features at the right side of the table.

In three aspects in Table **?**, the proposed method is at level with the advanced methods. Both *RESTful* and socket-based connections between instances are possible as well as switching between the two at runtime. This feature is extremely useful for **async processing** where the requesting party can poll for status rather than keep an open socket waiting for the reply. *Hierarchical and ultimately P2P structures* can be supported by both classes of methods. However, the proposed method supports a higher degree of freedom, ultimately aiming at Heroku- or Docker-like massive scale-out applications [2] with non-trivial topologies generated by VNE optimizations [5]. Both classes of methods are suitable for *heterogenous environments*, however, Ibis still lists this as a *partially resolved* problem [15].

The unique – different from both traditional and advanced methods – features of the proposed method are as follows.

Learning curve is extremely sharp with a short period of time required to understand and adopt the 3-way scripting method. The complexity problem is also tackled by Ibis [15] but remains unsolved – in fact, it is argued in [14] that any distributed programming framework involves a slow learning curve as long as it involves a bulky framework that attempts to be as generic as possible.

The proposed method is **platformless** in addition to being completely autonomous. Designs in [14] also intend to be platformless but lack the flexibility of the 3-way scripting proposed in this paper. In fact, the wireless spaces in [14] are much more uniform in terms of hardware function than the environments found in clouds. Several examples of such environments were discussed earlier in this paper.

Finally, the **traffic encoding** feature refers to the fact that both traditional and advanced methods encode the traffic exchange, both in terms of using a complex protocol and encrypting the payload. The proposal has a very lightweight protocol – only enough to provide the generic interface for all three modes in the 3-way scripts – however in terms of encryption the output is in plaintext/JSON but can be encrypted in user code if necessary.

5. Conclusion

This paper proposed a new class of distributed methods referred to as *distributed code* and implemented using the *3-way scripting* technique. There might be other techniques that would satisfy the three main objectives formulated in this paper – *volatility, heterogeneity and autonomy* – but the 3-way scripts appear to satisfy these objectives to a much higher degree than existing platforms. Note that the objectives originate from an existing research on distributed programming in wireless spaces, where this paper shows that there are many similarities between wireless spaces and cloud environments.

The *distributed code* method intends for the code to become the base unit in a scale-out cloud population. Just like in Heroku or Docker, such populations are best when built from a large number of relatively small and simple units of code. Existing literature also refers to such populations as *massively parallel* and/or *massively distributed*, the two terms tightly mixed in cloud environments. Although it may appear mutually exclusive, such environments can also be *heterogeneous*, especially in case of fog clouds where individual instances of code can encounter drastically different local environments across the cloud. The *distributed code* method through its *3-way scripts* offers enough flexibility to adapt to local specifics and even support non-trivial topologies when connecting to other instances in P2P mode. The simplest way to experience the flexibility of the proposed code is to switch its use from web API to local object mode and monitor the difference in achievable data throughput.

At present time, the presented 3-way scripting design is used in several working prototypes. The closest to traditional is the generic Docker-based applications running on large populations (copies) of the proposed code. The less traditional are the Big Data prototypes where the heterogeneity feature is tested to its fullest. Here, the proposed code is often run at the Big Data location in manual mode for debugging and monitoring purposes, always using the same 3-way script rather than relying on extra/external software.

Future work on the topic will focus on creating automation scripts (not frameworks) for deployment, monitoring and managing large populations of 3-way scripts. Documentation will also be written on common usecases in which 3-way scripts can be useful.

References

[1] Github Public Repository for the 3-Way Scripting Project. [Online]. Available:
https://github.com/maratishe/3wayscripting (July 2015)

[2] M.Zhanikeev, “A Cloud Visitation Platform to Facilitate Cloud Federation and Fog Computing”, IEEE Computer, vol.(in processing), May 2015.

[3] M.Zhanikeev, “Streaming Algorithms for Big Data Processing on Multicore”, Big Data: Algorithms, Analytics, and Applications, CRC, 2015.

[4] M.Zhanikeev, “Methods and Algorithms for Fast Hashing in Data Streaming”, Cryptography: Algorithms and Implementations Using C++, CRC, 2014.

[5] M.Zhanikeev, “A New VNE Method for More Responsive Networking in Many-to-Many Groups”, 7th International Conference on Ubiquitous and Future Networks (ICUFN), July 2015.

[6] M.Zhanikeev, “Experiments with application throughput in a browser with full HTML5 support”, IEICE Communications Express, vol.2, no.5, pp.167–172, May 2013.

[7] W.Emmerich, V.Gruhn (editors), Engineering Distributed Objects. Proceedings of ICSE Workshop, 1999.

[8] Distributed Objects Programming Topics. Apple Objective-C Manual, 2007.

[9] R.Buyya, C.Vecchiola, S.Selvi, Mastering Cloud Computing: Foundations and Applications Programming. Elsevier, 2013.

[10] F.Baude, D.Caromel, M.Morel, “From Distributed Objects to Hierarchical Grid Components”, On the Move to Meaningful Internet Systems (CoopIS, DOA, and ODBASE), Springer LNCS vol.2888, pp.1226–1242, 2003.

[11] M.Albano, L.Ricci, L.Genovali, “Hierarchical P2P Overlays for DVE: An Additively Weighted Voronoi Based Approach”, International Conference on Ultra Modern Telecommunications (ICUMT), pp.1–8, 2009.

[12] K.Birman, J.Cantwell, D.Freedman, Q.Huang, P.Nikolov, K.Ostrowski, “Live Distributed Objects for Service Oriented Collaboration”, Technical Report, Cornell University, 2009.

[13] T.Zink, O.Haase, J.Wasch, M.Waldvogel, “P2P-RMI: Transparent Distribution of Remote Java Objects”, International Journal of Computer Networks and Communications (IJCNC), vol.4, no.5, pp.17–34, 2012.

[14] J.Dedecker, T.Cutsem, S.Mostinckx, T.DHondt, W.Meuter, “Ambient-Oriented Programming in AmbientTalk”, 20th European Conference on Object-Oriented Programming (ECOOP), pp.230–254, 2006.

[15] M.Hajibaba, S.Gorgin, “A Review on Modern Distributed Computing Paradigms: Cloud Computing, Jungle Computing and Fog Computing”, Journal of Computing and Information Technology (CIT), vol.22, pp.69–84, 2014.