Journal of Information Processing Vol.24 No.1 99–108 (Jan. 2016)

[DOI: 10.2197/ipsjjip.24.99]

Regular Paper

Performance Management of Cloud Populations via Cloud Probing

Marat Zhanikeev[1,a)]

Received: January 1, 2015, Accepted: September 2, 2015

Abstract: Cloud population is a term that describes a cloud application distributed over many virtual machines or container-based boxes. Cloud platforms today offer simple tools for performance management (common example is load balancing) which are not sufficient for managing the performance of cloud populations. This paper proposes a new concept called cloud probing which is where applications themselves probe their host cloud platforms and optimize their own populations at runtime based on measurement data. This paper shows that even a simple optimization algorithm can lead to improved performance for the entire population. Since the only prerequisite function is the ability to migrate, the proposed method is also feasible in federated clouds where apps are fully in charge of managing their own populations spread across multiple cloud providers. This paper showcases the design of the TopoAPI that implements cloud probing, runs independently from physical platforms, and can therefore be used in federated environments.

Keywords: cloud probing, cloud populations, cloud performance management, active probing, live migration, topology optimization, migration cost, federated clouds

1. Introduction

Cloud Population is a recent concept hat describes distributed populations of resources in clouds implementing a single service [1]. Resources can be based on Virtual Machines (VMs) or containers – Heroku [25] or Docker [24] are the currently popular examples of the latter kind. Cloud populations are part of *cloud economy* where most interactions are based on 3-party contracts [2]. There are several existing examples of cloud populations. For example, a cloud-based video streaming service (CDN: content delivery service) needs a population of video sources and 3-party contracts for effective management of Quality of Service (QoS) [1].

Note that while the notion of cloud populations is discussed in literature, this paper is the first known attempt to define the technology for *managing performance* in such populations. To optimize performance, the proposal in this paper relies on the concept of *migration* ubiquitous in clouds today. The novelty of the proposal is in the part of *stress-based optimization* which uses probing data to model stress and migrations as unit actions for stress minimization. The two methods for *stress visualization* are also original in this paper, while, as the discussion further in this paper shows, traditional methods commonly rely on graph-based structures. The package containing all the novel concepts, optimization formulations, models and visualizations, and practical algorithms presented in this paper are referred to as **cloud probing**.

In terms of existing practice, clouds nowadays do not implement the notion of cloud populations. Instead, most clouds today offer a limited toolkit for performance management of a relatively small number of items, with a small range of practical usecases. For example, the popular Amazon cloud (AWS: Amazon Web Services) offers only basic tools like load balancing [23].

Figure 1 showcases the notion of cloud populations via the real measurement trace used throughout this paper. The measurement was implemented in the following settings. 15 VMs were created and spread randomly across 8 AWS regions – the number of VMs did not change during the entire experiment but VMs were constantly migrating between regions. Every 30 minutes, 5 VMs would migrate, each to the least populated area at the time, which created a roughly equal distribution of VMs across areas, with a constant rotation of the few areas with only one VM. During each 30 minute session, VMs would repeatedly send active probes to each other. The size of each probe was selected randomly from a set of values between 1 kbytes to 5 Mbytes. Direction of the probe was also selected randomly between GET (download) or POST (upload) requests – emulating web traffic

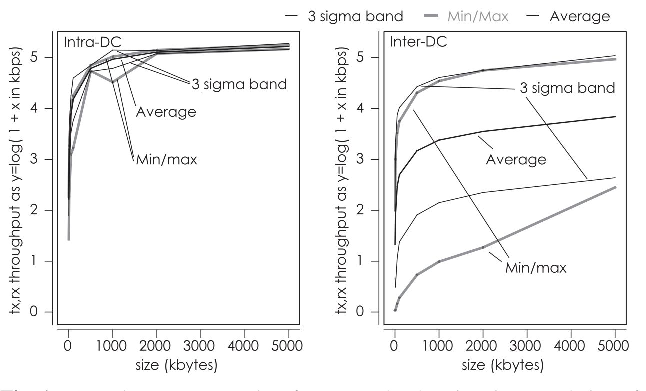


Fig. 1 Actual measurement data from a randomly migrating population of 15 VMs in the Amazon cloud.

[1] Department of Artificial Intelligence, Computer Science and Systems Engineering, Kyushu Institute of Technology, Iizuka, Fukuoka 820–8502, Japan

a) maratishe@gmail.com

between VMs. The experiment ran for several weeks until having collected more than 100 samples for each combination of regions per each hour of the day.

This collection of measurement data is referred to as *the trace* throughout this paper. Figure 1 also reveals the practical problems which can be resolved by the proposed method of *cloud probing*. Based on the setting, the following classification of measurement results is obvious. There is intra- and inter-DC *throughput*, which can be calculated from large bulks (3–5 Mbytes). When the bulk size is small (1-10 kbytes), we get intra- and inter-DC delay. These are rough definitions, but they fit into the overall theory of active probing [3].

Figure 1 does not make (for now) the distinction between specific areas or pairs of areas, and aggregates all the relevant results into intra-DC and inter-DC plots, showing *average* throughput plus *absolute min*, *max* and $\pm 3\sigma$. Average delay (calculated from throughput for small bulk size) is not drastically different between inter- and intra-DC and is relatively small for both. Throughput is better for intra-DC (local networking), but upper margins (absolute max and 3σ) are roughly the same for both.

The practical lessons learnt from Fig. 1 are as follows. Intra-DC is optimal in terms of performance but unfeasible in practice because a cloud-based CDN needs a geographically rich population [1], which means using multiple DCs is a practical necessity. In terms of delay, there is not much difference between the two cases, so, apps in both intra- and inter-DC settings will interact (delay is important for interactive apps) roughly at the same level of efficiency. In terms of throughput, the wide spread of inter-DC performance points to the obvious fact that different combinations of DCs that are selected to host parts of a given population should result in very different performance.

The above practical observations – especially the one about throughput on various combinations of DCs – can be used to formulate the following core problem resolved by *cloud probing*. The rest of this section formulates the problem, names the main players and presents the original solutions offered in this paper.

Since the Service Provider (SP) is in charge of its own population while Cloud Provider (CP) only supplies regionally diverse DCs and ability to migrate between them, SP can optimize the performance of its own population by measuring the current performance and triggering migrations in order to improve it. For a more detailed description of these players as part of the *cloud resource economy* refer to Ref. [1]. This paper refers to this technology as **cloud probing**. The ultimate runtime objective of cloud probing is to achieve an optimal distribution of VMs (AWS) or containers (Heroku, Docker) across the available space. The concept of *performance* is defined in specific metrics later in this paper.

The specific contributions of this paper are as follows:

- *cloud population*, *cloud probing* and the related notions are properly formulated expanding on the existing literature on cloud economy [1], [2], while the *cloud probing* technology is the original contribution in this paper;
- the generic model for performance management of cloud populations is formulated using state modeling and relation between performance improvement and migration cost – this contribution is not original but is required as the background for the optimization problem which is original;
- an original optimization framework for cloud probing is proposed along with the concepts of stress optimization and stress-centric visualization methodology;
- analysis is performed in form of a trace-based emulation using two example application scenarios and the proposed performance optimization;
- finally, the TopoAPI is proposed as an example implementation of cloud probing in practice – the API is also an original contribution of this paper.

2. The Concept of Cloud Populations

This section establishes the necessary background for the technology of performance optimization in cloud populations presented further in this paper.

Today, there are three main types of environments that can support cloud populations. Note that the naming in the classification below is arbitrary. However, the traditional *aaS formulations fail to describe some of these types which is why a new naming is offered.

Cloud Platforms describe environments like AWS [23]. Clients get raw access – they can create VMs, freely migrate them across the available regional coverage, etc. On the other hand, only basic performance management tools are offer, where load-balancing is the most common tool. Given the raw level of access, populations are not directly supported.

App Platforms describe environments like Heroku [25]. The notion of population is intrinsic to such environments, where *Heroku* specifically is built around the concept of scale which means that all the apps (short for applications) are multi-item populations by design. Clearly, such environments implement population management. However, it is important to remember that such environments do not manage *your population*. The management is performed on the entire set of resources in a given environment, without making a distinction of a sub-population for a given particular cloud service. For, example, Heroku runs 100% on top of AWS and one of its management objectives is to minimize the number of VMs by destroying unused web processes – the metric for *unused* is normally implemented as a timeout. This overall objective often conflicts with performance objectives of individual services, where the letter prefer to maintain a steady number of web processes and to not have to wait for them to start on demand. *App Platforms* do not offer tools for managing performance of populations at the grain of a specific (your own) app. To be specific, changing the scale of your applications (an option in Heroku) does not fall under the definition of *performance management*, as is shown further in this paper.

DIY Platforms (DIY: Do It Yourself) describe environments like Docker [24]. The notion of populations is again intrinsic but you the client have to do everything by yourself. Docker helps with automation of installation, cross-platform compatibility, etc. Docker also helps with the bulk size of migrations. In literature this is referred to as *the greybox problem* – to minimize the volume of bulk transfer during migration [16], [17]. Docker achieves this by minimizing the bulk size of containers. Yet, the process

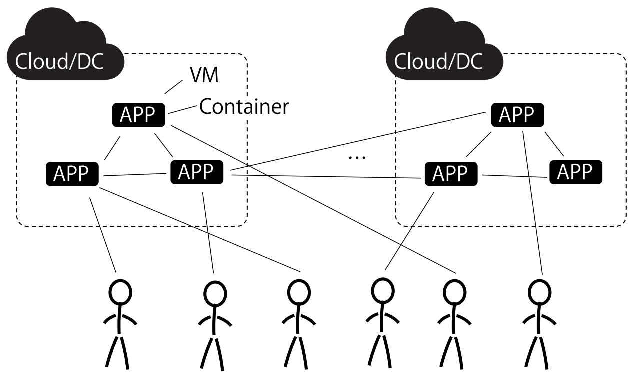


Fig. 2 Generic model of a cloud population.

of management is mostly manual, where one has to install and run Docker tools on each new VM prior to being able to run the *docker* command or write automation on top of it.

In practice, the *cloud probing* technology proposed in this paper can be implemented as a DIY population on top on one or more cloud environments. However, with some effort it can work on App Platforms. This topic is revisited in Section 9 when a practical implementation of the proposal is presented.

As a separate taxonomy from the platforms above, **Fig. 2** describes a *federated cloud* environment [1], [2] with multiple Cloud Providers (CPs) each operating multiple Data Centers (DCs). The main player – the Service Provider (SP) – runs a service on top of CP/DC infrastructure which consists of multiple apps, where one app can be one VM (in AWS [23]) or one container (in Heroku [25] or Docker ([24]). In case of container-based apps, unit of virtualization of cloud resources goes one level deeper as it is common to have one VM host multiple apps running in containers. The connection between Fig. 2 and the earlier taxonomy of platforms is that Fig. 2 shows that cloud federation is a natural environment for large-scale cloud populations, while each cloud in the federation can implement one or more platforms. The SP will have to approach each distinct platform differently but there are no practical limitations for SPs running on multi-platform cloud federations.

Figure 2 shows a non-trivial graph with both intra- and inter-DC links connecting individual apps. Finally, there is a geographically spread community of *end users* (EUs), end-to-end (e2e) QoS to which can be measured and considered as part of the overall performance of SP's population [8]. Management of EU community is also possible in practice via request forwarding (commonly used in CDNs [1]) where one can control distribution of EUs per app (per DC, per CP, and so on). For simplicity, this paper will not incorporate EU community in performance optimization, instead performing the analysis within the trace that contains real measurements across Amazon DCs.

3. Related Work

The notion of conducting active measurements for performance management is not new – the *active probing* technology has existed for several years [3] with a range of applications starting from primitive metrics like e2e *delay*, *throughput*, and *available bandwidth*, to such advanced uses as *tomography* and *network coordination* [3], [4]. Active probing is distinct from *passive monitoring* which is mostly conducted via packet traffic analysis. Passive monitoring also has a range of practical uses, from packet and flow classification [3], to e2e QoS classification for user communities (o2m and m2m patterns) [8], to the advanced high-performance methods that apply multicore hardware to online analysis of high-rate packet traffic [7]. Specifically in clouds, passive monitoring is part of the topic of *VM workload classification* [5] and a more generic methodology for inferring VM performance from traffic and other workloads [6].

Note that VM workload specifically and passive monitoring in general are not directly related to the topic of cloud population management. The reason is the same as was mentioned above while describing Heroku – passive monitoring can be used for managing the aggregate bulk of CP's resources but not at the grain of a subset of resources allocated for individual SPs. The difference here is crucial, the problem is fundamental and rooted in complexity of the respective optimization problems [9]. From this point on, this paper assumes that *population management* refers to different technologies in CP and SP forms, where *cloud probing* is part of the SP's population management technology.

CP population management is described by several methods in literature. The traditional methods is referred to as *VM placement* [10], [11], where the optimization problem is known as *bin packing* [21]. Since VM placement causes too many migrations when run in online/continuous form, *energy efficient migration* method can be used to maximize migration efficiency by accounting for *migration cost* [9]. Some methods focus on specific costs, for example proposing network-aware migration efficiency [12].

A large subset of this literature focuses on the migration cost itself, by measuring the actual cost [15], of minimizing the cost via so-called *greyboxes* (lighter bulk) [16], [17]. BigData networking – faster bulk transfer for BigData (VM images, containers, etc.) can also help lower the cost of migration, where one practical technology discussed in literature is *e2e circuit emulation* [20]. Note that Docker [24] can be viewed as a practical implementation of a greybox – the containers in Docker are optimized for size and are known to be much smaller than their traditional (non-optimized) versions.

Restating the distinction, CP population management *is distinct* from SP population management. CPs trying to optimize individual SP populations is unfeasible in practice [9] due to the complexity problem. Optimization of heterogeneous CP populations is already known to be an NP-hard problem [9]. Further splitting such populations into individual per-SP subsets would further increase the complexity. The take-home lesson here is that SP-side cloud population management is a practical necessity.

Cloud population management (in its intended SP form) is also distinct from *network coordination* where the technology is known under the names of *tomography* or *delay space clustering* [3], [4] in literature. Network coordination is suitable for highly distributed networks of nodes which is common in, for example, P2P streaming [1]. By contrast, in clouds SP normally *knows* the locations of its CPs/DCs/VMs/etc. Moreover, complex graphs connecting apps are rare in practice, which is why tomography in such environments would be an unnecessary excess. This paper shows further on that a simpler ring model is more feasible in practice, compared to a complex graph.

In terms of similarities, *cloud probing* can be viewed as *the reversed VNE* (Virtual Network Embedding) problem [18], [19]. In VNE, one performs the mapping of multiple virtual graphs onto a (single) shared physical graph. The target usecase in VNE literature overlaps with cloud probing – it is commonly presented as a technology that CPs use to optimize topologies of multiple SPs. VNE today are not found in practice in its full intended form, but partially supported primitive VNEs exist, for example, as private virtual networking in existing clouds AWS [23].

There is a subtle but obvious difference between *cloud probing* and VNE. While VNE *is* a technology for mapping, cloud probing assumes that the initial mapping (apps to VMs, etc.) has already been completed, after which cloud probing assumes that SP continuously measures the performance of its population and uses the results to optimize the initial mapping. In other words, VNE is a one-time static method with full knowledge of underlying and overlaying topologies, while *cloud probing* is a runtime dynamic optimization method. Repeating the earlier statement, VNE for a multi-SP mapping has higher complexity than the known NP-hard formulations [9].

Cloud probing relates to *active probing* and *network edge* in one package. Active probing is a practical technology for network edge – for example, the *homebox* project installs an active probe at users' homes for e2e active probing [13]. Network edge is also the ultimate target for clouds – this new kind of cloud is referred to as *fog computing* [14]. A new kind of platform – the Visitation Platform – is necessary to allow for VMs/containers to visit with a small cloud platform installed at end users' locations [14]. When SPs have populations based on heterogeneous hybrids of DC and *fog* infrastructures, *cloud probing* becomes a requirement for performance management of such populations.

4. The Basic Proposal

This section presents the proposal in general terms. The two main concepts presented in this section are *state model* and *incremental improvement diagram*. Both are conventional notions but are used in this paper in a non-conventional way. Specifically, the state model is based on individual migrations, but the state itself represents the performance of the entire population. The concept of incremental improvement of performance is based on the conventional *low-start improvement* notion but this paper is the first known attempt to incrementally improve performance of cloud populations via sequences of migrations with performance monitoring running in parallel. Section 3 had a detailed discussion of the difference between cloud populations and traditional resource management in clouds. For example, triggering multiple migrations for energy efficiency [9] uses a completely different toolkit and achieves a completely different optimizational objective.

4.1 State Model

Figure 3 depicts the generic concept of population management via probing (the word cloud is omitted for convenience). The state model describes the *groping by probing* process where each migration, generically speaking, has no guarantee that it will result in a better overall performance. In clouds today, there is no

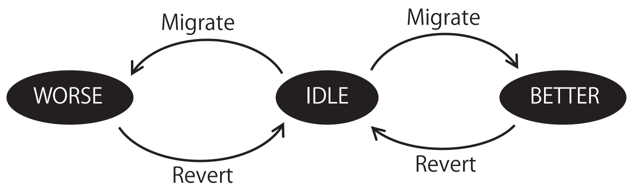


Fig. 3 State model for a cloud population that uses cloud probing to improve its performance.

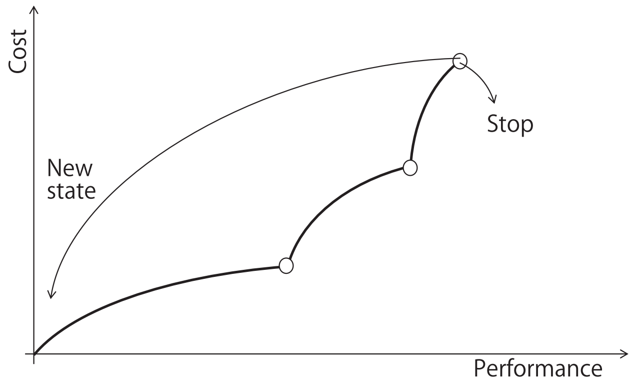


Fig. 4 The low-start nature of performance management based on cloud probing.

practical way to predict the outcome prior to migration. This rule will be upgraded further in this paper in a history-aware subset of methods.

The state model in Fig. 3 can be enhanced. While the model shows that we can always revert to the previous state by undoing the last migration, in practice this might not always restore the earlier state of performance – new VMs will be mapped to different racks, buildings, DCs in seemingly the same region. Moreover, there are time (daily, weekly, etc.) fluctuations in performance. To include this factor, the state model can be enhanced by defining, in addition to the *revert* transition, another *migrate* transition to yet a new state. Such a chain can continue infinitely in each direction. The model can also take into consideration the history of past migrations by assigning probabilities to transitions – such a model becomes a standard Markov chain. History-aware methods are placed out of scope of this paper but will be explored in the advanced cloud probing methods in future publications.

4.2 Low-Start Incremental Improvement

Partly taking into consideration the infinite chains, **Fig. 4** serves as a visual guide for the cost-vs-performance tradeoff. Migration always incurs cost divided into tangible values like billing for bulk transfer over the network, VM-shutdown/VM-start cycles in AWS [23] and less tangible estimates like the time required for bulk transfer (several minutes for AWS) and repeated migrate-revert or migrate-migrate cycles. Many specific definitions of migration cost can be found in literature [15].

Figure 4 shows that, regardless of the specific definition of cost, performance optimization will always follow the so-called *low-start* trend, which means that early improvements (from a random state) are easy and bring large performance improvements at a relatively small cost, while each further improvement comes at a higher cost and brings a smaller improvement margin. The trend is resolved by stopping the optimization process. The trend can also resolve naturally if the overall conditions in the population change drastically.

As a side note, it can be stated that, in practice, the latter (natural) way of resolving performance deadlocks is more common. Emulations further in this paper will also show that major changes in state are naturally common, which means that the *new state* operator in Fig. 4 is rarely needed in practice. With this experience added, conditions met in practice can be more accurately represented in Fig. 4 as infinite chains of transitions with stationary average cost and performance increment metrics. However, the discussion of the general notion in Fig. 4 remains useful.

5. Proposed Optimization

This section presents the proposed optimization problem. The ring-based visualizations provide the visual form for optimizational objectives and are also part of this section.

5.1 Optimization Problem

Let us put v (will call it *key* later) an arbitrary performance metric. It is always pairwise between nodes (CPs, DCs, etc.) a and b, i.e., v_{ab}. Same-node (intra-DC) v_{aa} (always a) and directional $v_{ab} \neq v_{ba}$ values are possible. Generically we have a graph $G(N, M)$ of n nodes and m links. With cloud probing, when SP attempts to implement active probing for all-to-all (a2a) combinations, m approaches n^2. The graph is always virtual, same as in VNE [18], because it is based on logical connections across apps.

For stress optimization, we collect a set of values $\{v_{ab}\}$ for pairwise links between nodes $a,\ b \in G$ but define stress S for each node as an aggregate of its link values

$$S_a = f(v_{aa}, v_{ab}, v_{ac}, \dots, v_{ax}), \quad where\{a, b, c, \dots x\} \in G, \quad (1)$$

where $f()$ is an arbitrary aggregator function (sum, average, etc.).

The stress optimization is then defined as minimization of the overall stress:

$$minimize \quad \sum S_x, \quad x \in G, \quad (2)$$

which is a generic formulation, while specific formulations should include practical constraints.

5.2 Stress in Cloud Populations

Stress optimization is a known problem in *graph drawing* [22] where the practical objective is to draw a graph in the most esthetically pleasing way. However, the same approach is fully applicable to performance optimization simply by using v_{ab} values for stress.

Yet, the stress optimization in this paper is distinct from graph drawing. While graph drawing deals with arbitrarily large graphs, cloud populations are normally relatively simple. In fact, it is common to use completely flat populations where each app performs exactly the same job. Heroku here is a vivid example, where large-but-flat topologies are referred to as *scaling-out* – increasing the scale without increasing the complexity. Another example is a cloud-based CDN where all apps use the same backend storage while the streaming function is implemented via a flat layer of streaming sources [1].

The subject of *reversed VNE* can be revisited here as well. VNE operates with complex topologies like *backbone, hub and spokes* [19]. In cloud probing, we start with a flat population, perform measurements for a subset of v_{ab} combinations (see reduction methods in Ref. [4]), and, if necessary, we can then impose a complex connectivity structure on top of the flat topology based on the measurement data. It is entirely possible for a cloud probing technology to create hub-and-spokes topologies as final product.

5.3 Cloud Populations as Stress Rings

Given that the initial populations are flat, this paper proposes to use **stress rings** as a visualization tool. Note that the rings are both visualization structures and visual optimization tools as is shown further in this paper. Strictly speaking, rings are graphs (one-level all-to-all meshes). Moreover, ring-based visualizations reflect the reality formed by all-to-all probing and can facilitate quick decision-making in respect to which apps to migrate. This paper defines two kinds of rings.

Simple Ring define stress for each a based on many aa or ab measurements but the outcome is handled for each a independently from b. For example, if one transfers the same VM image from one b to several a regions (cache replication in CDN [1]), then b is irrelevant, while difference in performance across several as is key to performance management.

Complex Ring defines stress for each a based on many ab measurements, where $a \neq b$. Each individual v_{ab} can then be decomposed so that performance for each b can be revealed and used for future optimization. To showcase the practical necessity, the above example can be slightly altered by specifying that VM images can migrate from multiple bs (it was one b before). In this case, some ab links can have higher stress then others.

5.4 Example Stress Rings from the Trace

Figure 5 shows several example visualizations based on *the trace* (data from the experiment at the beginning of this paper). The 3-tuple notation in plots is as follows. *Key* denotes a specific parameter selected to represent v. *Sizes* denotes the subset of bulk sizes selected for the visualization, in kbps. *Parties* specifies a filter (aa or ab) for measurement data, where aa means *only intra-DC* results. Visualizations use the actual symbolic names AWS uses to specify its 8 regions.

Rings themselves visualize stress, modeling it as an outside pressure on the surface of a balloon (ring). The higher the stress (pressure), the more a point on the ring sinks closer to the center of the ring. Note that absolute values are not important for ring visualizations while relative values have the obvious practical value – one can quickly identify the most stressed point on the ring and trigger migrations to rectify the respective performance problem. All values in Fig. 5 are averages of all values for all VMs for the respective region. *Bullet size* corresponds to variance across each set of values – in practical terms it represents *volatility* of stress.

The top-down reading of Fig. 5 is as follows.

The 1st (topmost) ring visualizes stress for bulk transfer of VM images (copy AMI in AWS terminology [23]). One region (Saopaulo) is drastically different, another (Virginia) is the next worse but not as drastic. Performance of this ring can be

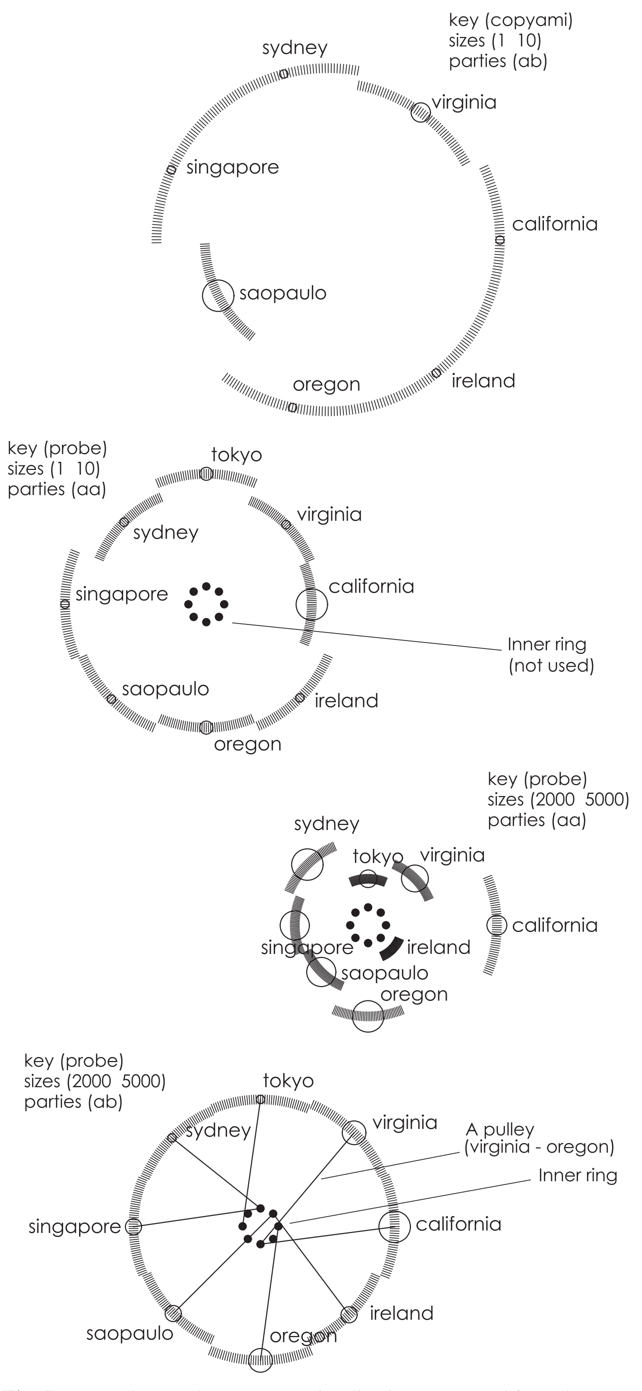


Fig. 5 Several example *stress ring* visualizations generated from the trace.

improved by removing Saopaulo from the population.

The 2nd ring visualizes stress for intra-DC interactive (small-bulk) populations. Note that the entire population can be spread across multiple regions, but performance management in this case focuses on the parts which have to interact in intra-DC groups. Tokyo is the least busy while California and Sydney are the busiest regions. Note that this ring is a Complex Ring but there are no pulleys in this specific ring because all *ab* parties are filtered out. This population can be improved by migrating interactive groups to less busy areas such as Tokyo, Ireland, Singapore, etc.

The 3rd ring is similar to the 2nd ring in that it visualizes intra-DC performance but this time the key is bulk transfer (large size). Ireland, Tokyo and Virginia are the worst and California and Oregon are the best environments for such groups. The performance improvement in this case is the same – app groups can migrate to better regions.

The 4th (bottom-most) ring is a showcase for the Complex Ring and visualizes inter-DC (*ab*) bulk transfer. The ring shows one pulley for each location on the outer ring in form of a link between a location on the outer ring with a location on the inner ring. This link is the visual representation of the *inner A pulls on outer B* relation between the locations. For example, Ireland is the most stressed region with Virginia contributing a bigger share of the stress. Another example is California whose overall stress is not drastically bad but its spread is large with Oregon contributing the most of it. To improve performance of this population, removing Oregon form California's remote parties would both improve the overall stress and lower variance for California's app group. Note that this ring shows the entire trace which has the smoothening effect on the data. Randomly selected subsets of the trace reveal more relative difference in emulations further in this paper. Also note that the concept of Complex Ring is not limited to the above practical formulations and can be based on any practical metric, depending on what a given SP considers important for the performance of its population.

The definition of stress in this section is used as is throughout the rest of this paper. All the practical optimization models and application scenarios in this paper assume that SP optimizes the performance of its population based on stress rings.

6. Evaluation Models and Scenarios

This section describes separately two models and two practical application scenarios (models) used later in this paper for analysis. Analysis always assumes a single SP that needs to manage performance of its own cloud population.

6.1 Application Scenarios

The two obvious scenarios are *Do Nothing* versus *Optimize*. The former refers to the existing state of affairs (traditional) where the concept of population performance management does not exist while the later (proposed) refers to performance optimization based on the stress rings and optimization problem described in the previous section. Optimization is always performed by triggering migrations on individual apps and monitoring changes in performance for yet further migrations.

6.2 Application Models

Each above scenario is applied to each of the following two example application models.

The Pooler model is about pooling large bulks of data via a distributed network of aggregators. One app (of many) is selected to be the final storage destination for the aggregated bulk and all other apps send their bulks to the selected destination. The following optimization logic is used (for the Optimize method). SP starts collecting data from *ab* probes that measure throughput (bulk transfer). Collection can be done at early station of bulk aggregation while migration of individual apps can be performed at a relatively low cost. Based on measurement data, the destination is selected as the app with the highest average throughput on connections to all other apps. All apps then send their bulks to the destination. The Simple Ring is sufficient to visualize stress for this model.

The Syncer model is the showcase for the Complex Ring (the bottom-most ring in Fig. 5 in the previous section) and implements a cloud-based CDN service [1]. The outer ring optimizes the *ab* stress for bulk transfer across data sources (see definition of sources in streaming in [1]) – which represents migration of sources across regions. The inner ring optimizes *ab* delay between sources *a* and end users *b*.

The role of pulleys is played by the following connection between the two rings implemented in the algorithm. In each optimization round in this model, SP first optimizes the router ring for bulk throughput and then the inner ring for delay. More specifically, this means that two migrations per cycle are triggered, the first migration optimizing the population for bulk throughput and the other attempting to improve the delay across members of the population by migrating the member with the largest average delay to all other members.

More details on how the models are emulated can be found in the next section.

6.3 Other Assumptions

Several assumptions are made. First, migration cost is considered to be negligible – the cost tradeoff explained earlier in this paper will be revisited in future publications. For example, in AWS, each shutdown/migrate/start cycle for VMs is charged 1 hour of VM's running time [23], which is a minor cost compared to the total running time of the entire application.

Secondly, while this paper discusses a simple version of cloud probing, a smarter version is not difficult to implement in practice. For example, it is easy for SP to keep a history of past measurements, which can be used to assign probabilities to the state model in Section 4.

The history can also be used directly for stress optimizations. For example, the immediately recent history of measurements can be used to decide which app to migrate. Stress optimization in this case can run continuously. Moreover, active probing itself does not need to be synthetic (dummy packets) but can be inferred from real application traffic. Future publications stemming from the core work in this paper will study the various cloud services for which cloud probing data can occur naturally as part of normal operation.

7. Evaluation Setup

As was mentioned above, the real measurement data used in this paper becomes *the trace* in trace-based emulation. This section provides more details on the trace and explains how emulations are conducted.

7.1 The Trace

There is always at least 100 values for each *aa* and *ab* combinations of regions for each hour of the day. Weekends are excluded from the trace to avoid the bias and retain focus on working days. All the days are merged into one working day. Multi-day emulations warp within this one day of the trace.

End user community is implemented as follows. The standard 2-peak distribution – peaking in mornings and evenings – of Internet use is used for all regions [9]. Note that the shape of the distribution is not important because the same shape is applied to all regions and therefore the main effect comes from the trace.

A community of 100 k users is emulated. E2e delay for each user is defined by selecting a random value from the trace values for 1 kbyte and 10 kbyte bulks. The assumption here is that users from a given region (close to the DC from that region) use an app running in another region (some are intra-region). 100 k users are allocated to regions based on the daily curve (Internet use), with the proper offset for each region from the GMT timezone. Emulation maintains a single value for time, but its value in each region is different depending on the offset from GMT.

7.2 One Emulation Run

One emulation run is executed as follows. A random time of day is selected as a starting point. 3 apps (VMs) are assigned to 3 distinct randomly selected regions. Emulation is then executed four times, once for each application model and each method, each time starting from the same initial conditions.

When emulation executes a bulk transfer (large size) or an interaction (small bulk) between apps or users and apps, a random value is selected from the set of values in the trace for that *ab* combination for that hour of the day. This way the emulation recreates statistically similar conditions to those which occurred during the probing experiment. Emulation time advances based on these values. However, some logical epochs are maintained. For example, one all-to-all probing session completes when the slowest bulk transfer ends.

7.3 Other Setup

The following application setup is applied. For the Pooler model, cloud probing session is executed in 50 all-to-all probing sessions using 5 Mbyte bulk probes. The application itself then executes two 100 Mbyte bulk transfers from two aggregator apps to the destination app. The Syncer model also executes 50 a2a probing sessions using Mbyte probes, but additionally, each end user sends 10 short (delay) probes to all three apps. Then, each user picks the closest of the three apps – representing perfect coordination and request forwarding during the main operation.

Note that while the above setting parameters were selected arbitrarily, they roughly represent the average app of each kind in practice. Specifically, description of the Syncer app is modeled after a cloud-based CDN [1].

A simple rule is applied to migrations. The same app can migrate at most 3 times but is not allowed to migrate for the next 3 rounds after each migration. This rule is a kind of volatility control which avoids the cases when only one app migrates throughout the entire emulation. More advanced logics will be considered in future publications.

8. Evaluation Results

This section presents evaluation results first by discussing randomly selection emulations in details and then discussing the overall results.

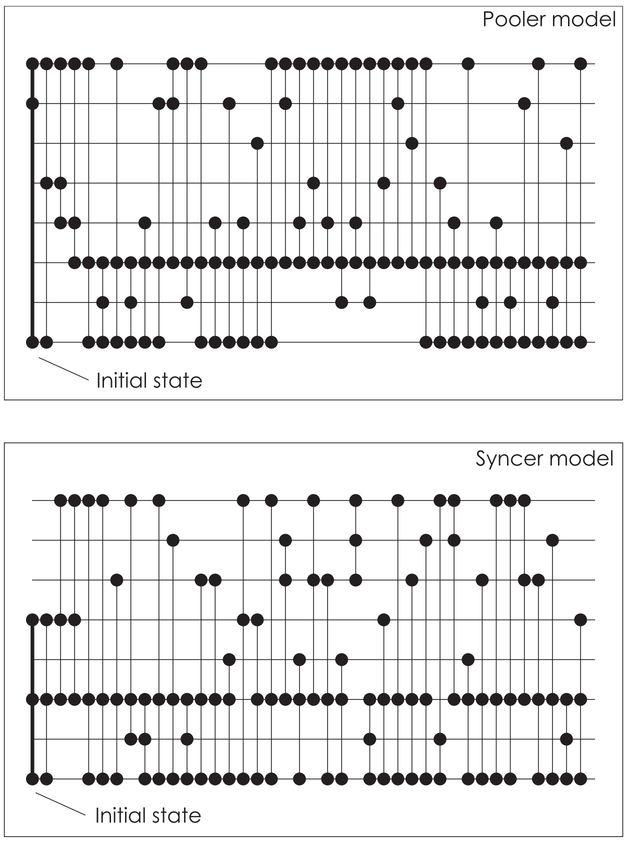


Fig. 6 Migration sequence for 3 VMs across 8 Amazon regions under *Pooler* (top) and *Syncer* (bottom) application models. XY scales have no values: X is the sequence of configurations and Y is the list of regions.

8.1 Analysis of Migration Schedules

Figure 6 shows example visualizations of migration schedules for both Pooler (top) and Syncer (bottom) models. The Pooler schedule is obviously more stable which is because it uses Simple Ring and therefore only one stress optimization per round. The Syncer schedule (Complex Ring) has two migrations per round and is more volatile. Regardless of the difference in volatility, both schedules show that some regions are preferred (1st, 6th, and 8th from the top) and occur more often than others. This means that some regions in the trace offer better performance on average than others. Note that the patterns are slightly different depending on whether the trace is mined for bulk (Pooler) or delay (Syncer) values.

8.2 Analysis of Overall Performance

Figure 7 shows the overall performance. All emulation results for each model are aggregated into one plot in form of distributions, thus allowing for visual comparison between *Do Nothing* and *Optimize* methods of dealing with population performance.

For the Pooler model (top), the completion time (of each application session in each emulation round) is the obvious practical metric. The chart shows that in more than 80% of cases, optimization results in better performance for the entire population. Note that the bulk for the application session is set to 100 Mbytes which is why the improvement is at the scale of several seconds. Larger bulks should result in bigger effect.

For the Syncer model (bottom), the practical performance

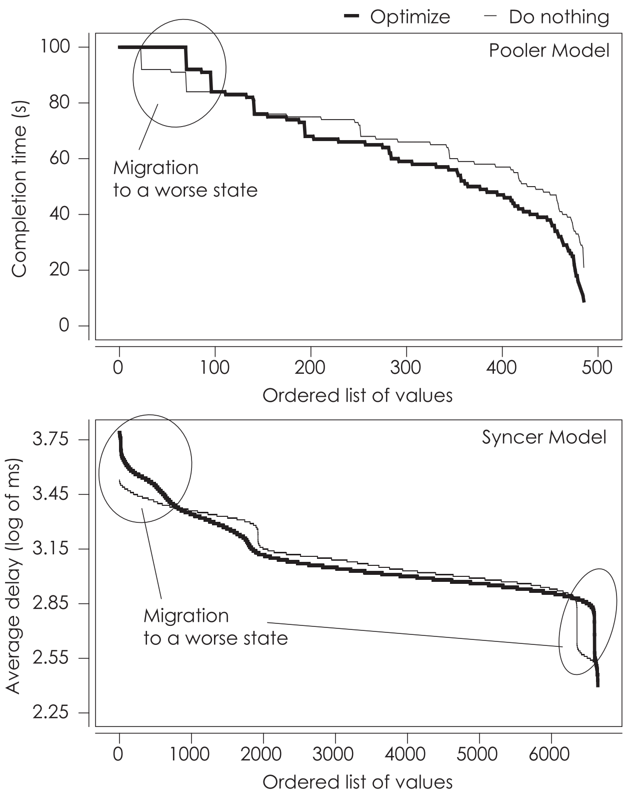


Fig. 7 Overall performance of *Optimize* (proposed) versus *Do Nothing* methods for the two example application models.

metric is the average e2e delay between users and their apps. Similarly, about 80% of cases result in a more optimal performance (vertical scale is in log). Note that the success of the Syncer model means that even 2-ring optimizations with different performance metrics can be successful.

9. Implementation

In federated clouds, independence of technologies from CPs and SPs is key. Existing examples in literature are, for example, resource metering conducted independently from resource providers [2]. In case of Cloud Probing API as a Service, it is also important to perform stress optimizations as independently as possible.

Figure 8 presents the TopoAPI – a generic API that implements cloud probing. There are two roles: Service Provider (SP) and TopoAPI itself. Prior to being able to use the API, SP has to have a contract with TopoAPI which comes with access tokens – traditionally using OAuth protocols same as in Ref. [2]. Once the contract is made and access tokens are exchanged, SP can start using the API.

One session of using TopoAPI goes as follows. SP starts a new session and gets an ID which it then can use to identify this session in later API calls. SP can then register any number of pairwise records using the *ADD* (*a*, *b*, *value*) API where *a*, *b* are parties and the nature of the performance metric *value* is up to SP.

When SP decides that sufficient volume of data has been collected, SP can complete the session by calling the *optimize* (*model*) API. The *model* here specifies a particular optimization method. The default is the stress optimization defined in this

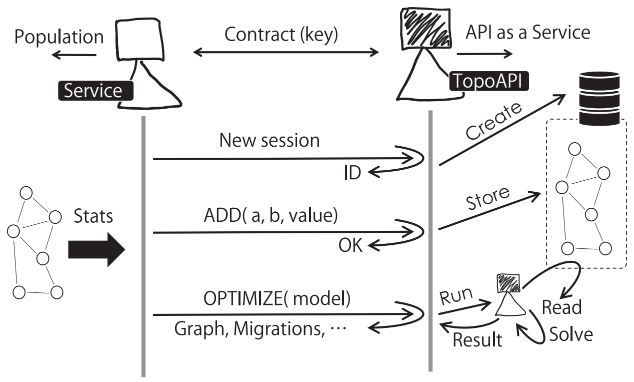


Fig. 8 The overall design of the *TopoAPI* – the practical implementation of the cloud probing concept proposed in this paper.

paper but future publications will look into other options. For example, TopoAPI may implement a set of methods defined for VNE in future versions [19].

Note that TopoAPI is platform-independent in two major ways. First, it is completely isolated from any CP or SP and can run in any 3rd-party environment. Secondly, TopoAPI is independent from the definition of performance. SP can register any metric with the API. As long as the metric is consistent throughout each API session, TopoAPI will successfully convert it into stress and generate migration recommendations that aim at stress minimization.

Also note that in existing clouds, the TopoAPI can be applied to both App Platforms and DIY Platforms from the taxonomy discussed in Section 2. It can be applied to Cloud Platforms automatically assuming that DIY Platforms (like Docker), in fact, provide the minimal environment necessary to run cloud populations in clouds. Although App Platforms (Heroku) do not normally provide a mechanism for services to trigger migration of individual apps, migrations can be emulated by changing the scale-out parameter. The practical advices for App Platforms will be provided in future publications on the topic.

10. Conclusion

This paper proposed a new technology called *cloud probing* which cloud service providers can use to optimize performance of their app populations. Cloud probing is not merely one of the available options for managing performance of cloud populations – it is the only feasible option available today. The only other alternative – global optimizations conducted by cloud providers for individual populations – is shown to be unfeasible in practice due to high complexity. Cloud probing reduces the complexity by allowing service providers to optimize their own populations while cloud providers facilitate the optimization by allowing apps migrate freely across regions. Many existing cloud platforms (like Amazon cloud) already fit this description and can be used by service providers (based on Docker, for example) to manage their populations. In fact, the concept of population is already common in clouds today, while the term population performance optimization is defined for the first time in this paper.

Cloud probing is named after its predecessor – the active probing. Like in traditional active probing, end-to-end network performance is inferred by sending dummy packet probes and measuring network response. Cloud probing is a large concept because active probing results are later used for performance optimization. The probes do not have to be dummy packets, instead probing data can be extracted from normal operation of a cloud population.

This paper formulated the concepts of stress and stress optimization. However, while stress optimization is commonly found in graphs (graph drawing, etc.), this paper shows that cloud populations do not require complex graphs. Instead, this paper proposed rings which can facilitate effective visualizations of stress. This paper proposed simple rings as well as complex structures with multiple rings.

Emulation in this paper was based on a trace created from real measurements conducted in Amazon cloud. This means that conditions in emulations were very close to those that occur for real cloud populations. Analysis was performed on two example application scenarios – Big Data pooling and cloud-based content delivery. The former depends on efficient bulk transfer across the populations while the later optimizes the 2-ring system where the first ring models the stress from bulk transfer across content delivery servers and the second ring models stress from end-to-end delay between end users and servers. For both cases, emulation showed that performance optimization can succeed for at least 80% of situations.

Future work on the subject of cloud probing is planned in several directions. Probing will be incorporated into normal/continuous operation of cloud populations where future work will study several existing popular cloud applications. The TopoAPI – the practical implementation of cloud probing – will be implemented as an actual web service and will be extended with optimization models used in the existing literature on virtual network embedding (VNE), thus, making the connection between the two topics. Finally, more work will be dedicated to software tools that support full automation of the various activities involved in cloud probing.

References

[1] Zhanikeev, M.: Multi-Source Stream Aggregation in the Cloud, *Advanced Content Delivery, Streaming, and Cloud Services*, Wiley (2014).

[2] Zhanikeev, M.: Coins in Cloud Drives Can Use OAuth for Micropayments and Resource Metering Alike, *9th International Conference on Future Internet Technologies* (*CFI*) (2014).

[3] Tanaka, Y. and Zhanikeev, M.: *Active Network Measurement: Theory, Methods, and Tools*, ITU Association of Japan, Tokyo (2009).

[4] Zhanikeev, M. and Tanaka, Y.: Application of Graph Theory to Clustering in Delay Space, *8th Asia-Pacific Symposium on Information and Telecommunication Technologies* (*APSITT*), Kuching, Sarawak, Malaysia, Paper No.B-7-2, pp.1–6 (2010).

[5] Andreolini, M., Casolari, S., Colajanni, M. and Messori, M.: Dynamic Load Management of Virtual Machines in a Cloud Architecture, *ICST CLOUDCOMP*, pp.201–214 (2009).

[6] Chandra, A., Gong, W. and Shenoy, P.: Dynamic Resource Allocation for Shared Data Centers using Online Measurements, *International Workshop on QoS* (*IWQoS*) (2003).

[7] Zhanikeev, M.: A Software Design and Algorithms for Multicore Capture in Data Center Forensics, *9th ACM Symposium on Information, Computer, and Communication Security Workshops* (*ASIACCS/SFCS*), pp.11–18 (2014).

[8] Zhanikeev, M.: A holistic community-based architecture for measuring end-to-end QoS at data centres, *Inderscience International Journal of Computational Science and Engineering* (*IJCSE*), Vol.10, No.3 (2015).

[9] Zhanikeev, M.: Optimizing Virtual Machine Migration for

Energy-Efficient Clouds, *IEICE Trans. Communications*, Vol.E97-B, No.2, pp.450–458 (2014).

[10] Dhiman, G., Marchetti, G. and Rosing, T.: vGreen: A System for Energy Efficient Computing in Virtualized Environments, *14th ACM/IEEE International Symposium on Low Power Electronics and Design*, pp.243–248 (2009).

[11] Xu, J. and Fortes, J.: Multi-objective Virtual Machine Placement in Virtualized Data Center Environments, *IEEE/ACM International Conference on Green Computing and Communications* (*GreenCom*) *jointly with Conference on Cyber, Physical and Social Computing* (*CPSCom*), pp.179–188 (2010).

[12] Stage, A. and Setzer, T.: Network-Aware Migration Control and Scheduling of Differentiated Virtual Machine Workloads, *CLOUD*, pp.9–14 (2009).

[13] Zhanikeev, M.: A Home Gateway Box with Meter, Probe and L2 QoS Policy Edge, *IEEE Conference on Computers, Software and Applications* (*COMPSAC*), pp.550–555 (2013).

[14] Zhanikeev, M.: A Cloud Visitation Platform for Federated Services at Network Edge, *10th Annual International Joint Conferences on Computer, Information, Systems Sciences, and Engineering* (*CISSE*) (2014).

[15] Voorsluys, W., Broberg, J., Venugopal, S. and Buyya, R.: Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation, *CloudCom*, pp.254–265 (2009).

[16] Wood, T., Shenoy, P., Venkataramani, A. and Yousif, M.: Black-Box and Gray-Box Strategies for Virtual Machine Migration, *4th USENIX Symp. on Networked Systems Design and Implementation*, pp.229–242 (2007).

[17] Antonio, C., Tusa, F., Villari, M. and Puliofito, A.: Improving Virtual Machine Migration in Federated Cloud Environments, *2nd International Conference on Evolving Internet*, pp.61–67 (2010).

[18] Lu, J. and Turner, J.: Efficient Mapping of Virtual Networks onto a Shared Substrate, Technical Report No.WUSCE-2006-35, Washington University in St. Louis (2006).

[19] Houidi, I., Louati, W. and Zeghlache, D.: A Distributed Virtual Network Mapping Algorithm, *International Conference on Computers and Communications* (*ICC*), pp.5634–5641 (2008).

[20] Zhanikeev, M.: Circuit Emulation for Big Data Transfers in Clouds, *Networking for Big Data*, CRC (in print) (2015).

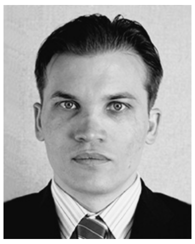
[21] Chekuri, C. and Khanna, S.: On Multidimensional Bin Packing Problems, *10th ACM Symposium on Discrete Algorithms*, pp.185–194 (1999).

[22] Kamada, T. and Kawai, S.: An algorithm for drawing general undirected graphs, *Inf. Process. Lett.*, Vol.31, No.1, pp.7–15 (1989).

[23] Amazon Web Services (online), available from ⟨http://aws.amazon.com⟩, (accessed 2014-12).

[24] Docker platform (online), available from ⟨https://www.docker.com⟩, (accessed 2014-12).

[25] Heroku (online), available from ⟨http://heroku.com⟩, (accessed 2014-12).



Marat Zhanikeev received M.S. and Ph.D. in Global Information and Telecommunications Studies from Waseda University in Tokyo, Japan, in 2003 and 2007, respectively. His research interests include network measurement, network monitoring, and network management, but also extend to practical applications related to these topics as well as non-traditional applications of information technology in general. He is presently an Associate Professor at Kyushu Institute of Technology (Kyutech), and is a Regular Member of IPSJ.