УДК 004.054
DOI: 10.18523/2617-3808.2021.4.88-92

Бенюх Л. І., Глибовець А. М.

РОЗРОБЛЕННЯ АРХІТЕКТУРИ СИСТЕМИ ПРОВЕДЕННЯ ВИСОКОНАВАНТАЖУВАЛЬНОГО ТЕСТУВАННЯ

У роботі проаналізовано основний інструментарій здійснення навантажувального тестування і тестування продуктивності, наведено приклади масштабування таких тестів і централізованої звітності метрик. Описано розроблену методологію та основні принципи побудови сучасної архітектури для ефективної реалізації підсистеми навантажувального тестування у безперервному постачанні коду.

Ключові слова: навантажувальне тестування, тестування продуктивності, Kubernetes, масштабування тестів, великонавантажувальні системи, централізована звітність результатів тестування, CI/CD, безперервне постачання коду.

Вступ

Важливість тестування продуктивності програмної системи важко переоцінити. Значно правильніше буде говорити про вчасність цієї активності. Практично будь-яка цифрова система, побудована за сучасними підходами та технологіями, може працювати без жодних критичних проблем із власною продуктивністю. Водночас, для будь-якої системи, особливо якщо вона стає популярною, з великою ймовірністю може настати такий момент, коли вона буде не спроможна впоратись із навантаженням, що постійно зростає, та стане нестійкою. Зокрема, за даними досліджень компанії Gartner [10], середня вартість 1 хвилини падіння та недоступності ресурсу може коштувати 5 тис. умовних одиниць, тобто 300 тис. у. о. за хвилину. Інше дослідження показало, що для 98 % великих інтернет-ресурсів вартість такої однієї години становитиме більше ніж 100 тис. у. о. [8]. Ба більше, для 33 % цих ресурсів вартість падіння буде від 1 до 5 млн у. о.

Тим не менше, більшість компаній, які розробляють і підтримують власні цифрові рішення, – від вебсайтів до будь-яких інших цифрових систем – часто сфокусовані переважно на функціоналі цієї системи та його відповідності вимогам, а не на продуктивності системи загалом. Такі наміри є досить природними, адже система має правильно виконувати очікувані від неї функції. Коли компанії починають стикатися з проблемами продуктивності, вони намагаються якнайшвидше не оптимізувати роботу програмного забезпечення, а додати більше потужностей – вертикальне та горизонтальне масштабування. Ця стратегія спрацьовує, але має обмеження. Адже додавання додаткових ресурсів не може бути нескінченним і рано чи пізно стикається або з самою архітектурою системи, або з можливостями самої компанії тощо.

Саме тому рекомендовано здійснювати навантажувальне тестування заздалегідь, планувати для цього час та ресурси, щоб мати достатньо часу на виправлення помилок, та в цілому розуміти межі системи. Водночас, для того щоб організувати повноцінне навантажувальне тестування, потрібні підготовлені спеціалісти, інструменти та інфраструктура, особливо коли ми говоримо про велике навантаження.

Вимоги до системи

Щоб почати розроблення дистрибутивної складної системи для здійснення високого навантаження, розглянемо спочатку традиційні підходи.

Як видно з рис. 1, для здійснення базового тесту достатньо локального персонального комп’ютера чи навіть ноутбука та одного з інструментів для навантажувального тестування. Треба зазначити, що цей підхід дає змогу генерувати порівняно невелике навантаження, від 50 до максимум близько 1000 віртуальних користувачів. Причому чим складніший сценарій самого тесту, тим менше користувачів можна згенерувати з однієї машини, адже самі інструменти для навантаження «жадібні» до оперативної пам’яті машин.

Ситуацію можна значно покращити, використавши досить сильний великий сервер чи взагалі віртуальну машину одного з хмарних

© *Бенюх Л. І., Глибовець А. М.,* 2021

Рис. 1. Процес здійснення базових навантажувальних тестів

провайдерів (AWS, Azure, GCP). Це дасть змогу здійснити значно більше навантаження, але знову-таки воно буде лімітоване однієї машиною. Рано чи пізно, цієї машини може знову стати недостатньо. До того ж такий підхід має недоліки:

– генерація навантаження відбуватиметься з однієї і тієї самої IP-адреси, що далеко від симуляції реальних користувачів;
– генерація також відбуватиметься з однієї і тієї самої географічної локації, що часто є проблемою для гео-масштабованих сервісів;
– сам підхід не є гнучким, адже він обмежений ресурсами однієї фізичної машини.

У цій роботі описано запропоновану архітектуру системи для здійснення ефективного розподіленого навантаження на високонавантажувальні сервіси.

Високорівневу архітектуру такої системи зображено на рис. 2.

Визначимо очікувані вимоги до такої системи:

1. *Можливість здійснювати навантаження майже необмеженого розміру.* Система має дати змогу здійснити навантаження 10, 50, 100 чи навіть більше одночасних віртуальних користувачів.
2. *Можливість здійснювати навантаження з різних географічних регіонів одночасно.* Це важливо для ресурсів, що використовуються не в одній країні, а в різних.
3. *Можливість динамічно збільшувати чи зменшувати навантаження під час здійснення самого тесту.* Це дасть змогу швидко реагувати на недостатнє чи, навпаки, велике навантаження. Зазвичай у таких випадках тест зупиняють і перезапускають з оновленою конфігурацією навантаження, що призводить до значних втрат часу, особливо якщо ми говоримо про довгі тести (2–8 годин). Система має давати змогу уникати подібних ситуацій.
4. *Централізоване збереження та відображення результатів.* Загалом, розгорнути дистрибутивну систему не так уже важко. Найбільша проблема – це якраз централізоване відображення результатів з усіх агентів (машин) навантаження. Часто безкоштовні інструменти обмежують у цьому (як Gatling), мотивуючи до використання платних ліцензій. Тому необхідно розробити підхід, який дасть змогу зберігати та відображати результати розподілених тестів централізовано, використовуючи при цьому інструменти з відкритим доступом до коду (open-source).
5. *Можливість відслідковування результатів тестів у реальному часі.* Такий підхід дасть можливість оперативно реагувати на результати тестів і змінювати стратегію навантаження у реальному часі, до повної зупинки тесту у випадку відмови сервісу під навантаженням.

Рис. 2. Високорівнева архітектура розподіленого навантаження

6. *Можливість систематичного та автоматизованого запуску тестів.* Система має давати можливість інтегруватися з різними CI/CD системами, такими як Jenkins, TeamCity, для вбудовування тестів у безперервну доставку коду.
7. *Система має бути недорогою в обслуговуванні.* Уся інфраструктура системи має динамічно збільшуватись чи зменшуватись до повної зупинки залежно від потреби у самій системі навантаження. У такий спосіб буде виправдано її використання на фоні уже присутніх на ринку сервісів (Blazemeter, Loadrunner, Gatling Frontline). Також із цієї вимоги випливає, що усі, або принаймні більшість, складових системи мають мати відкритий доступ до коду, або, іншими словами, «open-source» інструменти.
8. *Підтримка версій коду за принципом Tests as a Service.* Система має надавати змогу бути повністю інтегрованою у контроль та підтримку версій проєкту для відслідковування змін і збереження історії.

Враховуючи всі перелічені вище вимоги, ми можемо рекомендувати для побудови цієї системи інструменти, зазначені у таблиці.

Впроваджуючи поданий у таблиці набір технологій, ми покриваємо і виконуємо усі виставлені раніше вимоги. Єдиним джерелом затрат для побудови такої системи буде хостинг у одного з хмарних провайдерів. Усі інші інструменти або безкоштовні, або з відкритим кодом доступу, тобто теж безкоштовні.

Архітектура системи

Опишемо ключові інструменти та аргументуємо їх вибір.

Хмарний провайдер дасть нам змогу динамічно планувати та будувати інфраструктуру для навантажувальних тестів. Залежно від потреб, ми можемо розгорнути від 1 до практично безкінечної кількості віртуальних сервісів. Також ми можемо розгорнути їх у різних куточках планети, симулюючи навантажувальні тести з них.

Наступними запропонованими технологіями є Kubernetes [5] і Docker [2]. Kubernetes допоможе нам із менеджментом та оркеструванням віртуалізованих агентів із навантаження. До переваг Kubernetes, які допомагають нам задовольнити поставлені вимоги, можна віднести [9]:

– високу доступність інфраструктури;
– масштабованість і реплікацію;
– централізовану оркестровку та менеджмент репліками.

Gatling [3] було обрано як інструмент для написання тестів та здійснення безпосередньо навантаження, адже він дає змогу підійти до тестів як сервісу, інтегруючи сам навантажувальний проєкт у систему контролю версій кодом, наприклад GitHub. Мова, яка використовується в Gatling, – це Scala, тобто на базі JVM, що також дає таку перевагу, як багатоплатформність. Тобто тести можна запускати практично з будь-якої операційної системи. До того ж Gatling допоможе здійснити велике навантаження, використовуючи лише один агент, завдяки технології Akka [1], що дасть нам змогу використовувати менше машин ніж, наприклад, у випадку з JMeter, який є «жадібним» до оперативної пам'яті.

У Gatling є одне, але досить суттєве обмеження: хоча інструмент і має відкритий доступ до коду, тобто є безкоштовним, він не дає змоги здійснити розподілене навантаження, мотивуючи його користувачів використовувати платну версію Gatling Frontline [4]. Саме тут на допомо-

Таблиця 1

Рекомендований набір технологій для високонавантажувальної системи

Інструмент	Мета	Тип оплати	Покриття вимоги
Хмарний провайдер (AWS, Azure, GCP)	Хостинг та інші сервіси (реєстр, сховище)	Оплата за використані ресурси	#1, #2, #4
Kubernetes	Система для автоматизованого деплойменту, масштабування та менеджменту контейнеризованих систем	Open source	#3, #4, #7
Docker	Сервіс для віртуалізації систем	Open source	#3, #7
Gatling	Скриптинг і лоад-ранер	Open source	#7, #8
Logstash	Перетворення результатів для централізованого збереження	Open source	#4, #7
InfluxDB	База даних тимчасових рядів для збереження аналітичних даних	Open source	#4, #5, #7
Grafana	Візуалізація результатів	Open source	#5, #7
Telegraf	Агент для збирання інфраструктурних метрик	Open source	#7
Jenkins	Інструмент для збирання та запуску тестів	Open source	#6, #7
GitHub	Система контролю версій	Безкоштовно	#8, #7

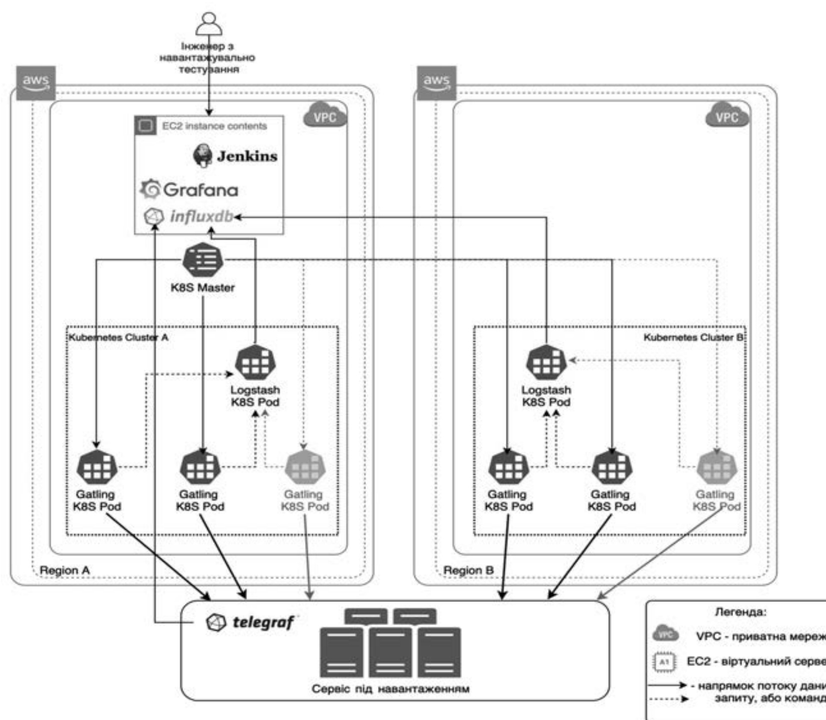

Рис. 3. Архітектура системи для здійснення розподіленого навантаження на прикладі хмарного провайдера AWS

гу нам приходить продукт з ELK набору технологій, а саме Logstash [6]. Останній допоможе нам трансформувати дані тестів з кожного агента так, щоб обійти обмеження Gatling та зберегти їх централізовано в базу даних тимчасових рядів.

Щодо бази даних, то одним із чудових рішень тут буде використання бази даних із тимчасовими рядами InfluxDB. Ця база якраз оптимізована для швидкого читання та запису аналітичних даних за часом, якими і є результати навантажувальних тестів [7].

База InfluxDB добре інтегрується з сервісом візуалізації даних Grafana – багатоплатформним вебдодатком для аналітики та інтерактивної візуалізації з можливістю побудови різного роду графіків і системи оповіщення. До того ж, у цей набір технологій чудово також інтегрується такий інструмент, як Telegraf, для збирання інфраструктурних метрик. Такий підхід дасть змогу візуалізувати одночасно результати навантажувальних тестів і метрик системи під навантаженням в одному місці – Grafana, що є зручно для аналізу результатів та пошуку взаємозв'язків.

Запропоновану архітектуру системи для здійснення розподіленого навантаження, на прикладі хмарного провайдера AWS та описаних вище технологій, зображено на рис. 3.

Висновки

Отримана в результаті архітектура може бути надалі реалізована як на приватних чи належних тому чи тому дата-центру серверах, так і з використанням віртуальних машин у хмарі. Завдяки інструменту кластеризації Kubernetes побудована архітектура є незалежною від інфраструктури її реалізації.

Слід зазначити, що одним із рекомендованих подальших кроків є безпосередня реалізація та випробування прототипу системи розподіленого навантаження, адже в процесі її побудови можуть виникнути практичні уточнення та деталі.

Загалом, пропонована архітектура дасть змогу побудувати підсистему для здійснення навантажувального тестування і тестування продуктивності з централізованим зберіганням і відображенням результатів, яка також інтегрована в систему CI/CD.

Список літератури

1. Akka [Electronic resource]. – Mode of access: https://github.com/akka/akka.
2. Docker [Electronic resource]. – Mode of access: https://www.docker.com.
3. Gatling [Electronic resource]. – Mode of access: https://gatling.io.
4. Gatling Frontline [Electronic resource]. – Mode of access: https://gatling.io/gatling-frontline.
5. Kubernetes [Electronic resource]. – Mode of access: https://kubernetes.io.
6. Logstash [Electronic resource]. – Mode of access: https://www.elastic.co/logstash.

7. Berman Daniel. InfluxDB vs. Elasticsearch for Time Series Analysis / Daniel Berman [Electronic resource]. – 2017. – Mode of access: https://logz.io/blog/influxdb-vs-elastic search.
8. Copeland Michael. The Cost of IT Downtime [Electronic resource] / Copeland Michael. – 2020. – Mode of access: https://www.the20.com/blog/the-cost-of-it-downtime/.
9. Kumar Atul. High Availability in Kubernetes | Detailed Guide [Electronic resource] / Atul Kumar. – Mode of access: https://k21academy.com/docker-kubernetes/high-availability-and-scalable-application-in-kubernetes.
10. Lerner Andrew. The Cost of Downtime [Electronic resource] / Andrew Lerner. – 2014. – Mode of access: https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/.

References

Berman, Daniel. (2017). *InfluxDB vs. Elasticsearch for Time Series Analysis.* Retrieved from https://logz.io/blog/influxdb-vs-elasticsearch.

Copeland, Michael. (2020). *The Cost of IT Downtime.* Retrieved from https://www.the20.com/blog/the-cost-of-it-downtime/.

Documentation of Akka. Retrieved from https://github.com/akka/akka.

Documentation of Docker Retrieved from https://www.docker.com.

Documentation of Gatling Retrieved from https://gatling.io.

Documentation of Gatling Frontline Retrieved from https://gatling.io/gatling-frontline.

Documentation of Kubernetes. Retrieved from https://kubernetes.io.

Documentation of Logstash. Retrieved from https://www.elastic.co/logstash.

Kumar, Atul. *High Availability in Kubernetes | Detailed Guide.* Retrieved from https://k21academy.com/docker-kubernetes/high-availability-and-scalable-application-in-kubernetes.

Lerner, Andrew. (2014). *The Cost of Downtime*. Retrieved from https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/.

L. Beniukh, A. Hlybovets

DEVELOPMENT OF THE ARCHITECTURE OF THE SYSTEM OF HIGH-LOAD TESTING

Testing system performance and its importance at the same time is difficult to overestimate or underestimate. It would be much more correct to talk about the timeliness of this activity. Virtually any digital system built on modern approaches and technologies can work without any critical problems with its own performance. At the same time, for any system, especially when it becomes popular, it is very likely that there will be a time when it will not be able to cope with the ever-increasing load and become unstable. However, most companies that develop and maintain their own digital solutions – from websites to any other digital systems – often focus primarily on the functionality of the system and its compliance, rather than on the performance of the system as a whole. Such intentions are quite natural, because the system must properly perform the functions expected of it. When companies start to face performance problems, they try not to optimize the software as soon as possible, but to add more capacity – vertical and horizontal scaling. This strategy works, but it has limitations. After all, the addition of additional resources cannot be endless and sooner or later rests either on the architecture of the system, or in the capabilities of the company itself, and so on.

Therefore it is recommended to carry out stress testing in advance, plan time and resources to have enough time to correct errors, and generally understand the boundaries of the system. At the same time, in order to organize full-fledged stress testing, trained specialists, tools and infrastructure are needed, especially when we are talking about heavy workload.

As part of this work, an analysis of various tools for the implementation of stress testing and performance testing, scaling of such tests and centralized reporting of metrics. As a result, approaches and principles for the construction of a modern architecture for the implementation of the load testing subsystem in the continuous supply of code were proposed.

Keywords: load testing, performance testing, Kubernetes, scaling tests, high-load system, centralized reporting of test results, CI / CD, continuous code delivery.

Матеріал надійшов 15.06.2021

Creative Commons Attribution 4.0 International License (CC BY 4.0)