Cognizant

Building a High Performance Reactive Microservices Architecture

As digital rapidly reshapes every aspect of business, IT organizations need to adopt new tools, techniques and methodologies. Reactive microservices are fast emerging as a compelling viable alternative.

Executive Summary

Growing demand for real-time data from mobile and connected devices is generating massive processing loads on enterprise systems, increasing both infrastructure and maintenance costs. Digital business is overturning conventional operating models, requiring IT organizations to quickly modernize their application architectures and update infrastructure strategies to meet ever-growing and ever-changing business demands.

To meet these elevated demands, the logical step is to re-architect applications from collaborating components in a monolithic setting to discreet and modular services that interact remotely with one another. This has led to the emergence and adoption of microservices architectures.

Microservices-based application architectures are composed of small, independently versioned and scalable, functionally-focused services that communicate with each other using standard protocols with well-defined interfaces. Blocking synchronous interaction between microservices, which is today's standard approach, may not be the optimum option for microservices invocation.

Reactive systems are flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change.

One of the key characteristics of the microservices architecture is technology diversity – and support for it. This extends to the programming languages in which individual services are written, including their approach toward implementing the capabilities embodied by microservices components within the reference architecture and the communication pattern between these components.

Systems or applications built on reactive principles are flexible, loosely-coupled, reliable and scalable. They are tolerant to failures and even when failures occur, they respond elegantly rather than with a catastrophic crash. Reactive systems are highly responsive, offering users a great user experience.

We believe that implementation of microservices architecture using reactive applications is a viable approach and offers unique possibilities. This white paper discusses aspects of reactive microservices architecture implementation and offers insights into how it helps to mitigate the growing demand for building scalable and resilient systems. It also illustrates a typical use case for microservices implementation, compares the performance of nonreactive and reactive implementation alternatives and, finally, concludes with a comparison that reveals how reactive microservices perform better than nonreactive microservices.

DEFINING MICROSERVICES ARCHITECTURE

Microservices architecture is an architectural style for developing software applications as a suite of small, autonomous services – optionally deployed on different tech stacks, and written in different programming languages – that work together running in its own specific process. The architecture communicates using a lightweight communication protocol, usually either HTTP request-response with resource APIs or light-weight messaging.

Microservices architectural styles[1] are best understood in comparison to the monolithic architectural style, an approach to application development where an entire application is deployed and scaled as a single deployable unit. In monoliths, business logics are packaged in a single bundle and they are run as a single process. These applications are scaled by running multiple instances horizontally.[2]

Reactive Systems

Reactive systems are flexible, loosely-coupled and scalable. This makes them easier to develop and amenable to change. They are significantly more tolerant of failures and, as mentioned above, they respond elegantly even when failures happen, rather than catastrophically failing. Reactive systems are highly responsive, offering users a great user experience.

The *Reactive Manifesto*[3] outlines characteristics of reactive systems based on four core principles:

- **Responsive:** Responds in a timely manner, provides rapid and consistent response, establishes reliable upper bounds and delivers a consistent quality of service.

- **Resilient:** Is responsive in the face of failure. Resilience is achieved though replication, containment, isolation and delegation.
- **Elastic:** Is responsive under varying workloads and can react to changes in request load by increasing or decreasing resources allocated to service these requests.
- **Message-driven:** Relies on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.

Reactive Microservices

Implementing microservices with reactive principles offers unique possibilities whereby each component of the microservices reference architecture, including the microservices components, are individually developed, released, deployed, scaled, updated and retired. It also infuses the required resilience into the system to avoid failure cascading, which keeps the system responsive even when failures occur. Moreover, the asynchronous communication facilitated by reactive systems copes with the interaction challenges as well as the load variations of modern systems.

Reactive microservices can be implemented using frameworks such as Node.js,[4] Qbit,[5] Spring Cloud Netflix,[6] etc. But we propose Vert.x[7] as the foundational toolkit for building reactive microservices.

IMPLEMENTING REACTIVE MICROSERVICES ARCHITECTURE VIA VERT.X

Vert.x is a toolkit to build distributed and reactive applications on the top of the Java Virtual Machine using an asynchronous nonblocking development model. Vert.x is designed to implement an Event Bus – a lightweight message broker – which enables different parts of the application to communicate in a nonblocking and thread-safe manner. Parts of it were modeled after the Actor methodology exhibited by Erlang[8] and Akka.[9] Vert.x is also designed to take full advantage of today's multi-core processors and highly concurrent programming demands.

As a toolkit, Vert.x can be used in many contexts – as a stand-alone application or embedded in another application. Figure 1 (next page) details the key features of Vert.x, revealing why it is ideally suited for microservices architecture.

Vert.x for Microservices Architecture Implementation

Vert.x offers various component to build reactive microservices-based applications. Getting there requires understanding of a few key Vert.x constructs:

- A Vert.x Verticle is a logical unit of code that can be deployed on a Vert.x instance. It is comparable to actors in the Actor Model. A typical Vert.x application will be composed of many

Vert.x Key Features

FEATURE	DESCRIPTION
Lightweight	Vert.x core is small (around 650kB in size) and lightweight in terms of distribution and runtime footprint. It can be entirely embeddable in existing applications.
Scalable	Vert.x can scale both horizontally and vertically. Vert.x can form clusters through Hazelcast[10] or JGroups.[11] Vert.x is capable of using all the CPUs of the machine, or cores in the CPU.
Polyglot	Vert.x can execute Java, JavaScript, Ruby, Python and Groovy. Vert.x components can seamlessly talk to each other through an event bus written in different languages.
Fast, Event-Driven and Non-blocking	None of the Vert.x APIs block threads; hence an application using Vert.x can handle a great deal of concurrency via a small number of threads. Vert.x provides specialized worker threads to handle blocking calls.
Modular	Vert.x runtime is divided into multiple components. The only components that can be used are those required and applicable for a particular implementation.
Unopinionated	Vert.x is not a restrictive framework or container and it does not advocate a correct way to write an application. Instead, Vert.x provides different modules and lets developers create their own applications.

Figure 1

Verticle instances in each Vert.x instance. The Verticles communicate with each other by sending messages over the Event Bus.

- A Vert.x Event Bus is a lightweight distributed messaging system that allows different parts of applications, or different applications and services (written in different languages) to communicate with each other in a loosely coupled way. The Event Bus supports publish-subscribe messaging, point-to-point messaging and request-response messaging. Verticles can send and listen to addresses on the Event Bus. An address is similar to a named channel. When a message is sent to a given address, all Verticles that listen on that address receive the message.

Figure 2 summarizes the key microservices requirements, the features provided by Vert.x to implement these requirements, and how they fit into our reference implementation.

Mapping Microservices Requirements to Vert.x Feature & Fitment with Our Reference Architecture

SERVICE COMMUNICATION	
Required Feature	Microservices interact with other microservices and exchange data with various operational and governance components in the ecosystem. Externally consumable services need to expose an API for consumption by authorized consumers. For externally consumable microservices, an edge server is required that all external traffic must pass through. The edge server can reuse the dynamic routing and load balancing capabilities based on the service discovery component described above. Internal communication between microservices and operational components or even between the microservices themselves may be over synch or asynchronous message exchange patterns.
Vert.x Support	Vert.x-Web may be used to implement server-side web applications, RESTful HTTP microservices, real-time (server push) web applications, etc. These features can be leveraged to create an edge server without introducing an external component to fulfill edge server requirements. Vert.x supports Async RPC on the (clustered) Event Bus. Since Vert.x can expose a service on the Event Bus through an address, the service methods can be called through Async RPC.
Reference Implementation	We have used the Vert.x-Web module to write our own edge server. The edge server may additionally implement API gateway features for API management and governance. We have leveraged the Async RPC feature to implement inter-service communication and for data exchange between microservices and operational components.

Figure 2 (continued on following page)

(Continued from previous page)

SERVICE SELF-REGISTRATION AND SERVICE DISCOVERY	
Required Feature	Tracking host and ports data of services with a fewer number of services is simple due to the lower services count and consequently lower change rate. However, a large number of modular microservices deployed independently of each other is a significantly more complex system landscape as these services will come and go on a continuous basis. Such rapid microservices configuration changes are hard to manage manually. Instead of manually tracking the deployed microservices and their hosts and ports information, we need service discovery functionality that enables microservices, through its API, to self-register to a central service registry on start-up. Every microservices uses the registry to discover dependent services.
Vert.x Support	Vert.x provides a service discovery component to publish and discover various resources. Vert.x provides support for several service types including Event Bus services (service proxies), HTTP endpoints, message sources and data sources. Vert.x also supports service discovery through Kubernetes,[12] Docker Links,[13] Consul[14] and Redis[15] back-end.
Reference Implementation	During start-up, microservices register themselves in a service registry that we have implemented using a distributed Hazelcast Map,[16] accessible through the Service Discovery component. In order to get high throughput, our implementation includes a client side caching for service proxies, thereby minimizing the latency added due to service discovery for each client-side request.
LOCATION TRANSPARENCY	
Required Feature	In a dynamic system landscape, where new services are added, new instances of existing services are provisioned and existing services are decommissioned or deprecated at a rapid rate. Service consumers need to be constantly aware of these deployment changes and service routing information which determines the physical location of a microservice at any given point in time. Manually updating the consumers with this information is time-consuming and error-prone. Given a service discovery function, routing components can use the discovery API to look up where the requested microservice is deployed and load balancing components can decide on which instance to route the request to if multiple instances are deployed for the requested service.
Vert.x Support	Vert.x exposes services on the Event Bus through an address. This feature provides location transparency. Any client systems with access to the Event Bus can call the service by name and be routed to the appropriate available instance.
Reference Implementation	We have exposed our microservices as Event Bus addresses whose methods can be called using Asynchronous RPC. This has a positive impact on the performance during inter-microservice communications as compared to synchronous HTTP/S calls between microservices.
LOAD BALANCING	
Required Feature	Given the service discovery function, and assuming that multiple instances of microservices are concurrently running, routing components can use the discovery API to look up where the requested microservices are deployed and load balancing components can decide which instance of the micro-service to route the request to.
Vert.x Support	Multiple instances of the same service deployed in a Vert.x cluster refer to the same address. Calls to the service therefore are automatically distributed among the instances. This is a powerful feature that provides built-in load balancing for service calls.
Reference Implementation	We have exposed our microservices as Event Bus addresses whose methods can be called using Asynchronous RPC. Load balancing is automatically achieved when multiple instances of the same service are concurrently running on different instances of the cluster.
CENTRALIZED CONFIGURATION MANAGEMENT	
Required Feature	With large numbers of microservices deployed on multiple instances on multiple servers, application-specific configuration becomes difficult to manage. Instead of a local configuration per deployed unit (i.e., microservice), centralized management of configuration is desirable from an operational standpoint.
Vert.x Support	Vert.x does not ship with a centralized configuration management server, like Netflix Archaius.[17] We can leverage distributed maps to keep centralized configuration information for the different microservices. Centralized information updates can be propagated to the consumer applications as Event Bus messages.
Reference Implementation	We have leveraged Hazelcast Distributed Map[18] to maintain the microservice application-related property and parameter values at one place. Any update to the map sends update events to applicable clients through the Event Bus.

Figure 2 (continued on following page)

(Continued from previous page)

HIGH AVAILABILITY, FAIL-OVER AND FAILURE MANAGEMENT	
Required Feature	Since the microservices are interconnected with each other, chains of failure in the system landscape must be avoided. If a microservice that a number of other microservices depends on fails, the dependent microservices may also start to fail, and so on. If not handled properly, large parts of the system landscape may be affected by a single failing microservice, resulting in a fragile system landscape.
Vert.x Support	Vert.x provides a circuit breaker component out of the box which helps avoid failure cascading. It also lets the application react and recover from failure states. The circuit breaker can be configured with a timeout, fallback on failure behavior and the maximum number of failure retries. This ensures that if a service goes down, the failure is handled in a predefined manner. Vert.x also supports automatic fail-over. If one instance dies, a back-up instance can automatically deploy and starts the modules deployed by the first instance. Vert.x optionally supports HA group and network partitions.
Reference Implementation	In our implementation, we have encapsulated the core service calls from the composite service through a circuit breaker pattern to avoid cascading failures.
MONITORING AND MANAGEMENT	
Required Feature	An appropriate monitoring and management tool kit is needed to keep track of the state of the microservice applications and the nodes on which they are deployed. With a large number of microservices, there are more potential failure points. Centralized analysis of the logs of the individual microservices, health monitoring of the services and the virtual machines on which they are hosted are all key to ensuring a stable systems landscape. Given that circuit breakers are in place, they can be monitored for status and to collect runtime statistics to assess the health of the system landscape and its current usage. This information can be collected and displayed on dashboards with possibilities for setting up automatic alarms against configurable thresholds.
Vert.x Support	Vert.x supports runtime metric collection (e.g., Dropwizard,[19] Hawkular,[20] etc.) of various core components and exposes these metrics as JMX or as events in Event Bus. Vert.x can be integrated with software like the ELK stack for centralized log analysis.
Reference Implementation	Our custom user interface implementation provides near-real-time monitoring data to the administrator by leveraging web sockets. Monitoring console features include: • Dynamic list showing the cluster nodes availability status. • Dynamic list showing all the deployed services and their status. • Memory and CPU utilization information from each node. • All the circuit breaker states currently active in the cluster. We used an Elastic Search, Logstash and Kibana (ELK) stack for centralized log analysis.
CALL TRACING	
Required Feature	From an operational standpoint, with numerous services deployed as independent processes communicating with each other over a network, components are required for tracing the service call chain across processes and hosts to precisely identify individual service performance.
Vert.x Support	Vert.x can be integrated with software like ZipKin for call tracing.
Reference Implementation	We have used ZipKin for call tracing.
SERVICE SECURITY	
Required Feature	To protect the exposed API services the OAuth 2.0 standard is recommended.
Vert.x Support	Vert.x supports OAuth 2.0, Shiro and JWT Auth, as well as authentication implementation backed by JDBC and MongoDB.
Reference Implementation	We have leveraged Vert.x OAuth 2.0 security module to protect our services exposed by the edge server.

Figure 2

IMPLEMENTATION: A USE CASE

The use case we considered for our reference implementation involves an externally consumable microservice which, when invoked, calls five independent microservices, composes the response from these services and returns the composite response to the caller. The composite service – FlightInfoService – is meant to be consumed by web portals and mobile apps and is designed to fetch information pertaining to a particular flight, including the flight schedule, current status, information on the airline, the specific aircraft, and weather information at the origin and destination airports.

Our Approach

- **Setup:** Composite and core services, including all operational governance components (for request routing, discovery, call tracing, configuration management, centralized log monitoring and security), were deployed in separate dedicated virtual machines as single Java processes.
- **Process:** Apache JMeter was used to invoke the FlightInfoService composite service. The three scenarios (Spring Cloud Netflix-Synch, Spring Cloud Netflix Asynch and Vert.x) were executed serially in the same environment with equivalent components running on the same virtual machines.
- **Configuration settings:** Composite and core services including all governance components were executed with their default configuration settings.

Our Technology Stack

We implemented this microservice ecosystem using two technology stacks, with each adopting different message exchange patterns.

- **We chose Spring Cloud Netflix for implementing the scenario where the composite service invokes the core services synchronously.** Spring Cloud Netflix provides Netflix OSS integrations for Spring Boot apps through auto-configuration and binding to the Spring Environment and other Spring programming model idioms. The stack includes implementations for service discovery (Eureka), circuit breaker (Hystrix), intelligent routing (Zuul) and client-side load balancing (Ribbon) (see Figure 3).

Spring Cloud Netflix Implementation

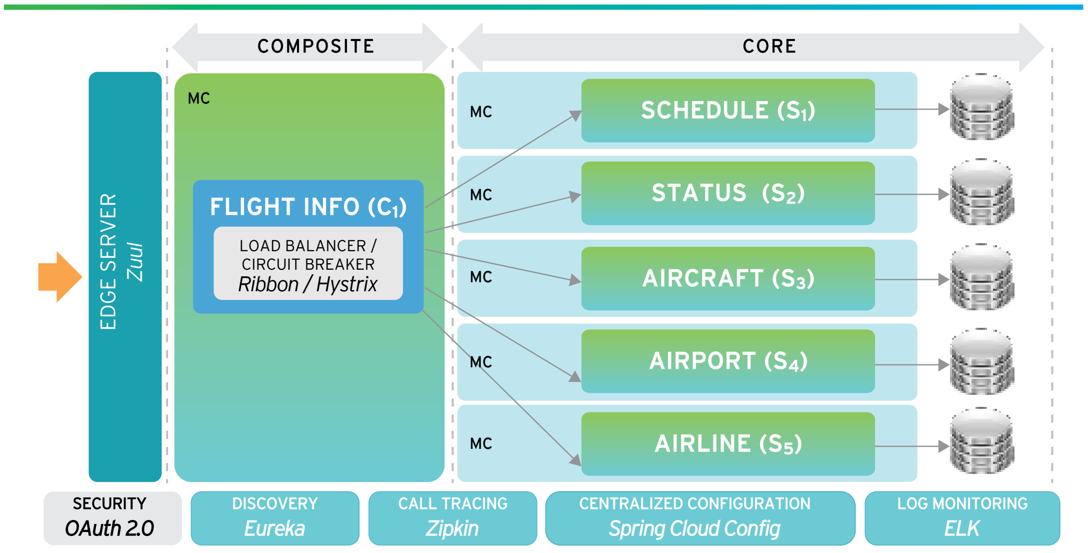

Figure 3

Our Vert.x Implementation

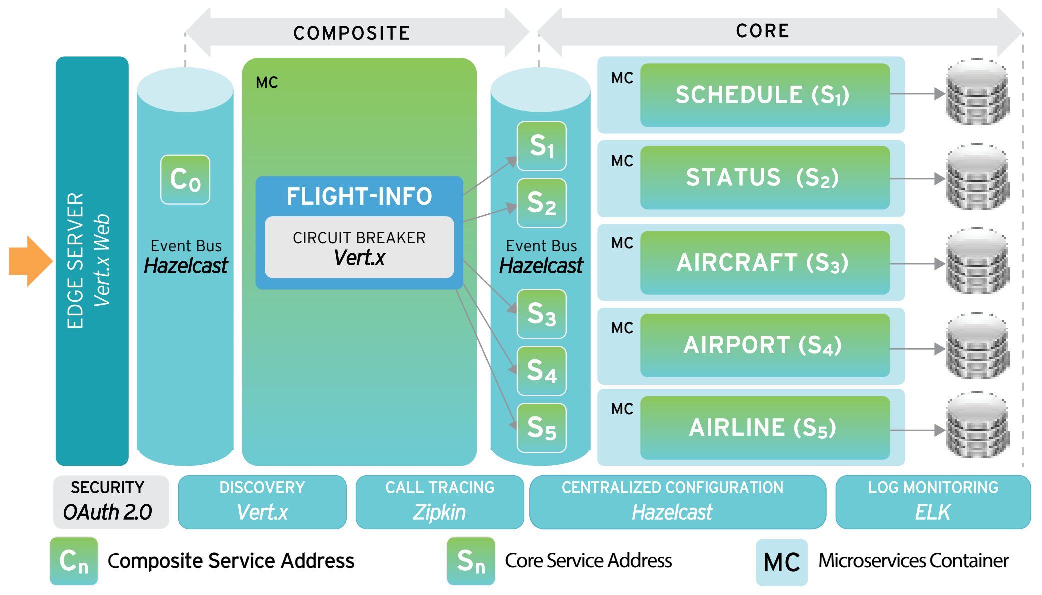

Figure 4

- **We used Vert.x for implementing the reactive and asynchronous message exchange pattern between composite and core services (see Figure 4).** The microservices were implemented as Vert.x Verticles. Details of how we implemented the microservices are discussed above.

Reactive microservice architecture implementation using Vert.x has the following characteristics, some of which are positively desirable for enterprise-scale microservices adoption:

- All components in the Vert.x microservice ecosystem (including the core and composite microservices and the different operational components) communicate with each other through the Event Bus implemented using a Hazelcast cluster. Internal service communication (i.e., between composite and core services) uses Async RPC, resulting in significant throughput improvement over synch HTTP, which is the default in Spring Cloud Netflix (see Figure 5, next page).
- Vert.x supports polyglot programming, implying that individual microservices can be written in different languages including Java, JavaScript, Groovy, Ruby and Ceylon.
- Vert.x provides automatic fail-over and load balancing out of the box.
- Vert.x is a toolkit enabling many components of a typical microservices reference architecture to be built. The microservices are deployed as Verticles.

GAUGING THE TEST RESULTS

Figure 5 shows the throughput comparison between Spring Cloud Netflix (Sync and Async-REST) (as per Figure 3) and the Vert.x implementation (as per Figure 4) in our lab environment.[21]

Observations

- **Spring Cloud Netflix-Sync implementation:** We observed that the number of requests processed per second was the lowest of the three scenarios. This can be explained by the fact that the server request thread pools were get-

Throughput Comparison

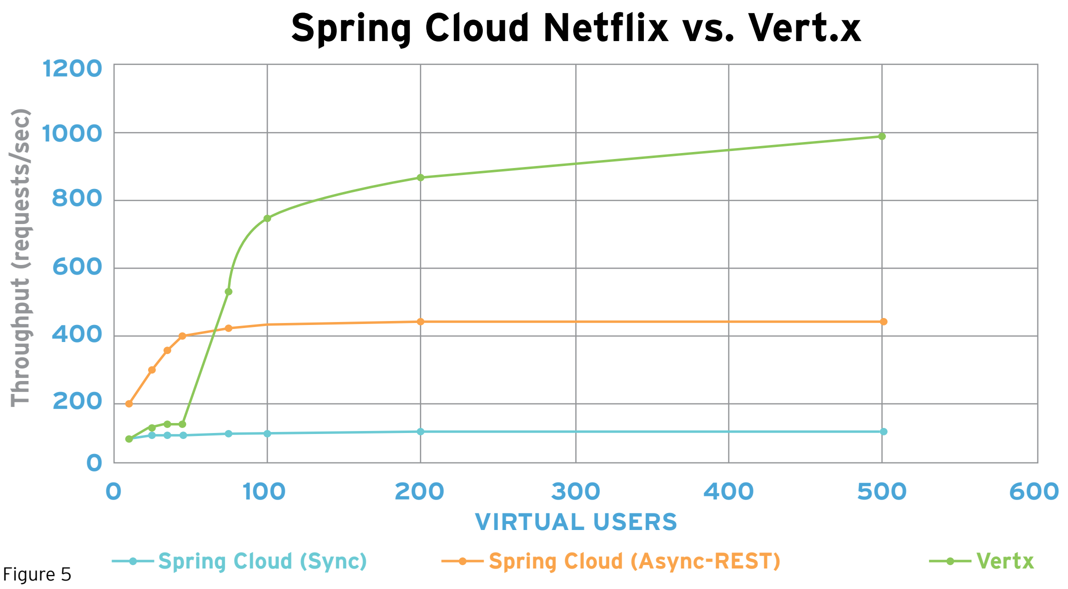

Figure 5

ting exhausted as the composite to core service invocations were synchronous and blocking. The processor and memory utilization were not significant. The core and composite services were blocking in I/O operations.

- **Spring Cloud Netflix-Async-REST implementation:** The Asynch API of the Spring Framework and RxJava help to increase the maximum requests processed per second. This is because the composite and core service implementations used the Spring Framework's alternative approach to asynchronous request processing.

- **Vert.x implementation:** We observed that the number of requests processed per second was the highest of the three scenarios. This is primarily due to Vert.x's nonblocking and event-driven nature. Vert.x uses an actor-based concurrency model inspired by Akka. Vert.x calls this the multi-reactor pattern which uses all available cores on the virtual machines and uses several event loops. The edge server is based on the Vert.x Web module which is also nonblocking. Composite to core service invocations use Async RPC instead of traditional synchronous HTTP, which results in high throughput. Even Vert.x JDBC clients use asynchronous API to interact with databases which are faster compared to blocking JDBC calls. Overall, Vert.x scales better with the increase of virtual users.

LOOKING FORWARD

Reactive principles are not new. They have been used, proven and hardened over many years. Reactive microservices effectively leverages these ideas with modern software and hardware platforms to deliver loosely-coupled, single-responsibility and autonomous microservices using asynchronous message passing for interservices communication.

Our research and use case implementation highlight the feasibility of adopting reactive microservices as a viable alternative to implement a microservices ecosystem for enterprise applications, with Vert.x emerging as a compelling reactive toolkit. Our work underscored the greater operational efficiencies that can be obtained from reactive microservices.

FOOTNOTES

[1] Microservices, http://www.martinfowler.com/articles/microservices.html

[2] Ibid.

[3] *The Reactive Manifesto*, http://www.reactivemanifesto.org/

[4] Node.js, https://nodejs.org/en/

[5] QBit, https://github.com/advantageous/qbit

[6] Spring Cloud Netflix, https://cloud.spring.io/spring-cloud-netflix/

[7] Vert.x, http://vertx.io/

[8] Erlang, https://www.erlang.org/

[9] Akka, http://akka.io/

[10] Hazelcast, https://hazelcast.org/

[11] JGroups, http://jgroups.org/

[12] Kubernetes, https://kubernetes.io/

[13] Docker Links, https://docs.docker.com/docker-cloud/apps/service-links/

[14] Consul, https://www.consul.io/

[15] Redis, https://redis.io/

[16] Hazelcast Map, http://docs.hazelcast.org/docs/3.5/manual/html/map.html

[17] Netflix Archaius, https://github.com/Netflix/archaius

[18] Hazelcast Distributed Map, http://docs.hazelcast.org/docs/3.5/manual/html/map.html

[19] Dropwizard, http://www.dropwizard.io/1.1.0/docs/

[20] Hawkular, https://www.hawkular.org/

[21] Load Generation Tool: JMeter (Single VM) [8 Core, 8 GB RAM]; Core Services: Each core service is deployed in a separate VM with 4 Core and 4 GB RAM; Composite Service: Deployed in a separate VM with 4 core and 16 GB RAM; Edge Server: Deployed in a separate VM with 4 core and 16 GB RAM; Operational Components: Deployed in separate VM with 4 core and 16 GB RAM.

ABOUT THE AUTHORS

Dipanjan Sengupta

Chief Architect, Software Engineering and Architecture Lab

Dipanjan Sengupta is a Chief Architect in the Software Engineering and Architecture Lab of Cognizant's Global Technology Office. He has extensive experience in service-oriented integration, integration of cloud-based and on-premises applications, API management and microservices-based architecture. Dipanjan has a post-graduate degree in engineering from IIT Kanpur. He can be reached at Dipanjan.Sengputa@cognizant.com.

Hitesh Bagchi

Principal Architect, Software Engineering and Architecture Lab

Hitesh Bagchi is a Principal Architect in the Software Engineering and Architecture Lab of Cognizant's Global Technology Office. He has extensive experience in service-oriented architecture, API management and microservices-based architecture, distributed application development, stream computing, cloud computing and big data. Hitesh has a B.Tech. in engineering from University of Calcutta. He can be reached at Hitesh.Bagchi@cognizant.com.

Satyajit Prasad Chainy

Senior Architect, Software Engineering and Architecture Lab

Satyajit Prasad Chainy is a Senior Architect in the Software Engineering and Architecture Lab of Cognizant's Global Technology Office. He has extensive experience with J2EE related technologies. Satyajit has a B.E. in computer science from University of Amaravati. He can be reached at Satyajitprasad.Chainy@cognizant.com.

Pijush Kanti Giri

Architect, Software Engineering and Architecture Lab

Pijush Kanti Giri is an Architect in the Software Engineering and Architecture Lab of Cognizant's Global Technology Office. He has extensive experience in Eclipse plugin architecture and developing Eclipse RCP applications, API management and microservices-based architecture. Pijush has a B.Tech. in computer science from University of Kalyani. He can be reached at Pijush-Kanti.Giri@cognizant.com.

ABOUT COGNIZANT

Cognizant

World Headquarters
500 Frank W. Burr Blvd.
Teaneck, NJ 07666 USA
Phone: +1 201 801 0233
Fax: +1 201 801 0243
Toll Free: +1 888 937 3277

European Headquarters
1 Kingdom Street
Paddington Central
London W2 6BD England
Phone: +44 (0) 20 7297 7600
Fax: +44 (0) 20 7121 0102

India Operations Headquarters
#5/535 Old Mahabalipuram Road
Okkiyam Pettai, Thoraipakkam
Chennai, 600 096 India
Phone: +91 (0) 44 4209 6000
Fax: +91 (0) 44 4209 6060

TL Codex 2654