Guaranteed Latency Applications in Edge-Cloud Environment

Petr Hnetynka
Petr Kubat
Rima Al-Ali
Ilias Gerostathopoulos
Danylo Khalyeyev
Modern cyber-physical systems

• Combine distributed embedded devices with computation in cloud

• Applications:
  ▪ Smart agriculture,
  ▪ Smart power grid,
  ▪ Smart traffic, ...

• Include thousands to millions of devices

• Inclusion of cloud makes possible advanced data analysis and decision-making
  ▪ making the system “smarter”
Real-time requirements

- Interaction with the physical world
  - Leads to the presence of the real-time requirements
    - e.g. a real-time video stream has to be processed without any significant delay

- Interaction with the cloud
  - Happens even inside the real-time tasks running on embedded devices
    - The cloud has to participate in overall real-time guarantees
      - No way to ensure these guarantees is provided by the cloud software
Existing approaches generally try to reduce the communication latencies

- By bringing the cloud closer to user
- By caching the data
- By prediction

All these approaches still work on best-effort

- i.e. no real-time guarantees provided
Our approach

- Combining edge-cloud processing with runtime performance awareness

- Real-time guarantees are provided by:
  - Pre-assessing the application
  - Monitoring the application performance
  - Predicting the application performance based on historical observations

- Cloud-centric control of applications
  - Traditional declarative deployments
    - Extended with timing requirements specification
Running example

- An augmented reality application
  - A mobile phone application
  - A processing part in the cloud
    - Analyzes the video stream from clients
    - Sends back the augmenting information
      - e.g. recognized faces

- The information has to be displayed with minimum delay
  - Even better, with a guaranteed communication latency
Structure of the approach

- **Q1: Specification of the real-time guarantees**
  - In line with existing practices
  - Establishing a verifiable contract

- **Q2: Assessment of cloud applications**
  - To determine whether the response time can be guaranteed
  - Performed automatically

- **Q3: Providing guarantees at runtime**
  - In face of changing conditions
    - Background load
    - User mobility
Q1: Real-time guarantees specification

- Traditional declarative specification of microservices
  - + Measurement probes specification
  - + Real-time requirements specification
- Probe – a function that performs a performance test
  - Does not take any inputs
  - Can be measured at runtime
  - Strongly correlates with the operation that needs to be guaranteed
- Timing requirements are specified over the probes
  - In contrast to real operations
Q1: Extended deployment descriptor

Our implementation extends Kubernetes deployment specification

- Contains a specification of timing requirements
  - “below X ms in Y% of cases”
- Defined over probes
  - Also a part of the extended descriptor

```yaml
kind: Deployment
metadata:
  name: recognizer-deployment
labels:
  app: recognizer
spec: # microservices specification
  template:
    metadata:
      labels:
        app: recognizer
    spec:
      containers:
        - name: recog
          image: d3srepo/recog
          ports:
            - containerPort: 7777
    probes: # probes
      - name: recognize
    timingRequirements: # timing requirements
      - name: recognize limit
        probe: recognize
        limits:
          - probability: 0.999
            time: 50 # Max. 50ms in 99.9% cases
          - probability: 0.99
            time: 30 # Max. 30ms in 99% cases
```
Q2: Assessment of an application

- Performed before the actual deployment of the application
- Verifies feasibility of the timing requirements
  - Informs the developer whether the application can be admitted and on what terms
- The probes are invoked many times in this process
  - Each invocation collects data about the probe’s behavior
Application performance is measured with different background workloads
- IO-intensive, CPU-intensive, memory-intensive, ...
- Gives us estimates on how different applications impact each other

System counters are collected in the process
- Instruction count, cache miss count, IOPS, ...
- Allows us to categorize probes by performed computation type

Gradually, this builds knowledge about application performance
- Used in what-if analysis about the impact of different applications on each other
  - The precision of the analysis grows with time
Q3: Providing the real-time guarantees

- To have a control over the deployment of applications, we provide the **Controller**
  - An intermediary between the user and the cloud
  - Collects information from the nodes via *node agents*
  - Controls the probe invocation via *application agents*
  - Responsible for execution of the self-adaptation loop
Q3: Self-adaptation loop

Building a constraint optimization problem

Finding a solution - a new deployment configuration

Comparing the desired configuration to the current one

Creating an execution plan - a set of actions that will bring the cloud to the desired state

Monitoring the performance of applications

Monitoring the state of the cloud

Analysis

Planning

Monitoring

Execution

Knowledge

Kubernetes Cloud

Redeploying the microservices according to the execution plan
Our approach provides statistical guarantees on the response time of the edge-cloud applications.

Timing requirements are specified directly as a part of the deployment specification.

Guarantees are kept in changing conditions:
- Thanks to performance awareness and adaptation.

Key ideas of the approach:
- Specification of the requirements over the pre-defined probes.
- Automatic pre-assessment of the application.
- Building a queryable knowledge model for improving adaptation decisions.
Our experiments show that applications can be successfully categorized based on resource utilization.

Currently, our framework includes:
- A control architecture over K8S
- Prototypes of all of its main components

Thanks!