Guest Lecture: Cloud Computing

CPEN400A - Building Modern Web Applications - Winter 2018-1

Julien Gascon-Samson

The Univerity of British Columbia
Department of Electrical and Computer Engineering
Vancouver, Canada

ece Electrical and Computer Engineering

Thursday November 15, 2018

Definitions

Guest Lecturer: Julien Gascon-Samson

- Post-Doctoral Fellow at UBC
  - PhD from McGill University (Montreal, 2017)
  - Master's in Computer Engineering (École Polytechnique de Montréal, 2011)
  - Undergrad in Software Engineering (École Polytechnique de Montréal, 2009)
- In Jan. 2019: will be appointed as Assistant Professor at ÉTS Montréal / University of Quebec (Software & IT Engineering Dept.)
  - On the lookout for highly-motivated MSc / PhD students – funded positions available :-)
- Research
  - Internet Of Things (IoT)
  - Cloud / Edge / Distributed Systems
  - Publish/Subscribe
  - Networking for Multiplayer Games

Distributed Computing

Distributed System: Definition (Wikipedia)

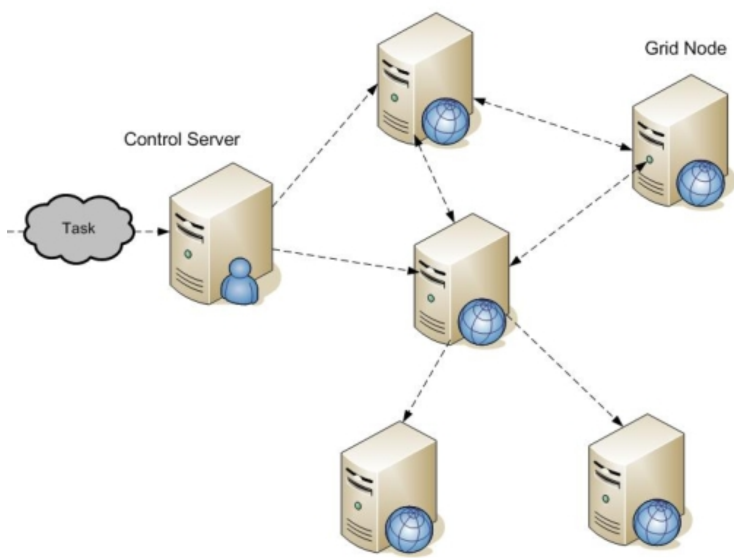
> *A system whose **components** are located on **different networked computers**, which then **communicate** and **coordinate their actions** by **passing messages** to one another.*

- Very broad! Different models are possible:
  - Centralized
  - Peer-to-peer
  - Hybrid

Grid Computing

Grid Computing: Definition (Wikipedia)

*A **combination** of **computer resources** from multiple administrative domains applied to a **common task**.*



Utility Computing

Utility Computing: Definition (Wikipedia)

The packaging of ***computing resources*** *(computation, storage etc.)* ***as a metered service*** *similar to a traditional public utility.*

Cloud Computing?

- Grid Computing + Utility Computing?
- Very hard to define – can mean so many different things to different parties!
- Many definitions

Cloud Computing: Definition (NIST)

Cloud computing is a model for enabling ***ubiquitous****,* ***convenient****,* ***on-demand network access*** *to a* ***shared pool of configurable computing resources*** *(e.g., networks, servers, storage, applications, and services) that can be* ***rapidly provisioned and released*** *with* ***minimal management effort or service provider interaction****. This cloud model is composed of five essential characteristics, three service models, and four deployment models.*

Cloud Characteristics

Cloud Characteristics (1)

1. On-demand Self Service

- Ability to provision computing capabilities without intervention
  - Computation (“aka machine”) time
  - Storage

Cloud Characteristics (1)

1. On-demand Self Service

- Ability to provision computing capabilities without intervention
  - Computation ("aka machine") time
  - Storage

2. Broad network access

- Capabilities available over the network
- Accessible by *thin* and *thick* clients (e.g., desktop/laptops, mobile devices, etc.)

Cloud Characteristics (2)

3. Resource pooling

- Multi-tenancy: the same cloud infrastructure can serve multiple customers, host multiple VMs, applications
- Computing resources are *pooled* (to serve multiple users)
  - Storage
  - Processing
  - Memory
  - Network
- *Physical* and *logical* resources are dynamically assigned and reassigned according to consumer demand
- Location independence
  - Precise location of the resources
  - Only a general idea (e.g., Amazon EC2 US-east)

Cloud Characteristics (3)

4. Rapid elasticity

- *Elastic* provisioning – scaling up and down
- Can be done automatically
- To consumers: pool of resources might appear to be infinite

Cloud Characteristics (3)

4. Rapid elasticity

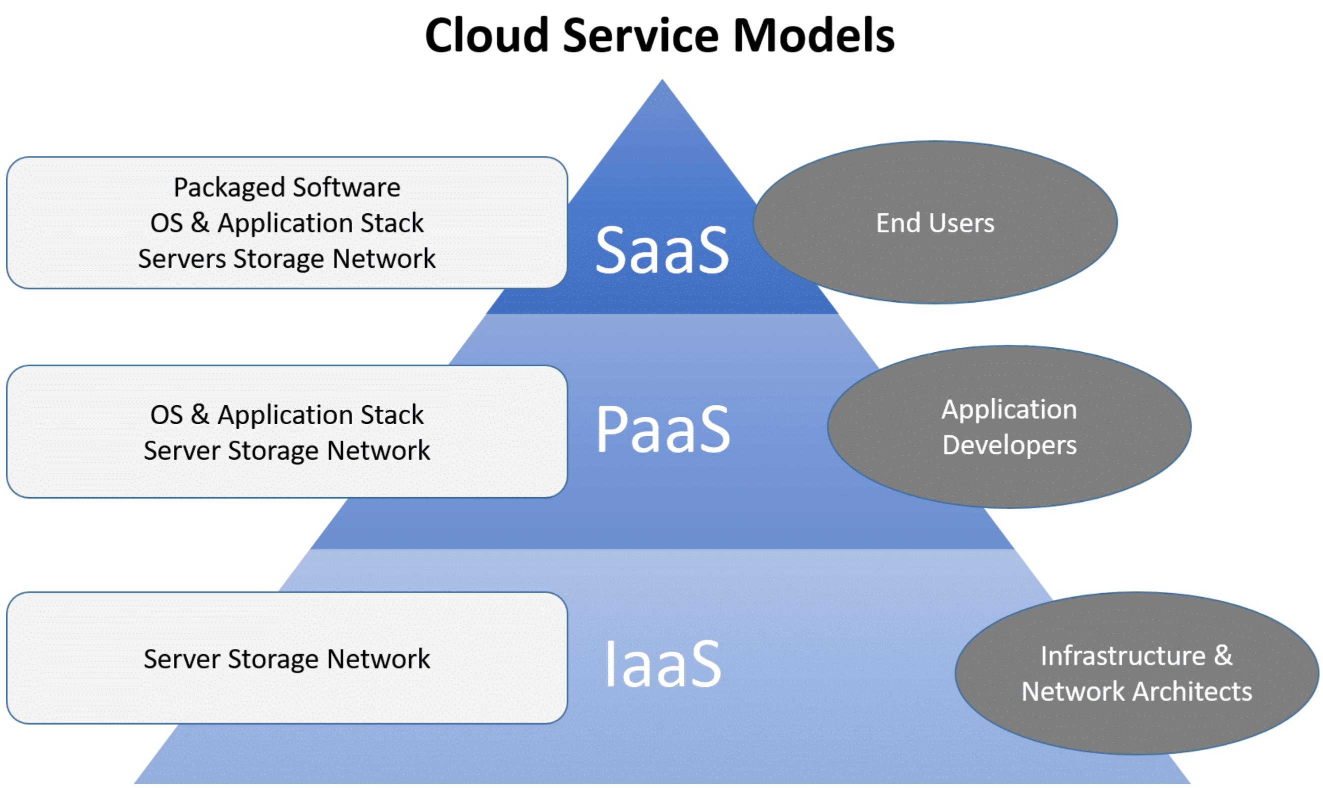
- *Elastic* provisioning – scaling up and down
- Can be done automatically
- To consumers: pool of resources might appear to be infinite

5. Measured service

- *Metering* of the different resources
  - CPU (e.g., $/CPU time in ms)
  - Network bandwidth (e.g., $/gb)
  - Processing (e.g., $/X requests)
  - Storage (e.g., $/gb)
- Monitoring, controlling, reporting
- Full transparency for cloud operator and consumer

Service Models

Hierarchy of Service Models (Source)



Infrastructure as a Service (IaaS)

- Consumer can provision virtualized computing resources (aka VMs)
    - Processing, storage, network, GPU
- Can include OS and applications, or be bare metal
- Example: Amazon EC2, Azure
- Consumer doesn't manage the hardware (physical or virtualized)
    - But has control over the OS, storage, applications, and limited network settings
        - e.g., firewall, port redirection, VLans, etc

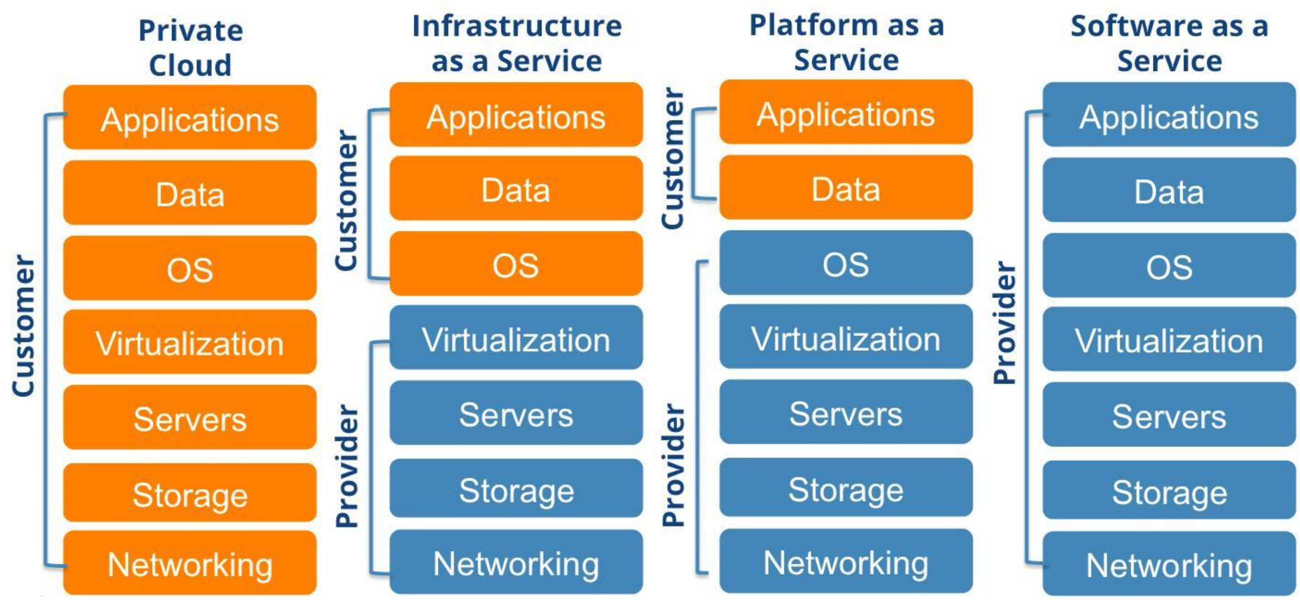
Platform as a Service (PaaS)

- Deployment onto cloud infrastructure (IaaS) of **consumer or acquired applications**
  - Written into a variety of languages
  - Using a variety of libraries, services, tools supported by the provider
  - e.g., Web apps (Heroku, Google App Engine)
- No control over underlying cloud infrastructure!
- Control over deployed applications
- Might have limited control over configuration settings of the hosting environment (e.g., Apache config files)

Software as a Service (SaaS)

- Use the provider's specific applications
  - Over the cloud provider's infrastructure (hardware + software)
- Accessible from various clients
  - Thin & thick clients, mobile, web (e.g., web-based email)
- Consumer does not manage the underlying cloud infrastructure (network, servers, OS, storage, applications)
- Exception: limited user-specific application configuration settings (e.g., GMail settings)

Provisioning in Service Models (Source)



Class Activity – Which service model?

1. Editing a document online on Google Docs
2. Testing a new Linux Kernel on an Amazon VM
3. Purchasing gifts on Amazon
4. Accessing a MySQL database service
5. Deploying a Python application
6. Provisioning a virtual machine

Class Activity – Which service model? – Solution

1. Editing a document online on Google Docs ⇒ **SaaS**
2. Testing a new Linux Kernel on an Amazon VM ⇒ **IaaS**
3. Purchasing gifts on Amazon ⇒ **SaaS**
4. Accessing a MySQL database service ⇒ **PaaS or SaaS**
5. Deploying a Python application ⇒ **SaaS**
6. Provisioning a virtual machine ⇒ **IaaS**

Deployment Models

Public cloud

- Open use by general public
- Owned by business, academic, government organization, or a combination
- Exists on premise of cloud provider
- Example: Amazon, Google, MS Azure

Private cloud

- Exclusive use of a single organization with multiple consumers
  - e.g. business units
- Owned, managed, operated by organization, or a third-party, or a combination
- May exist on or off premises
- Example: A large company (e.g., Amazon Internal Cloud)

Community cloud

- Exclusive use of a specific community of consumers from **organizations with shared concerns**
  - Mission, security requirements, policy, compliance considerations
- Owned, managed, operated by one or more organizations in the community, a third party, or a combination of them
- May exist on or off premises
- Examples: Amazon Government Cloud, clouds that comply with BC data policies (e.g., UBC Workspace)

Hybrid cloud

- A composition of two or more distinct cloud infrastructure

Virtualization

What is virtualization?

- Decoupling from the physical computing resources (physical hardware)
- Cloud provider might have heterogeneous hardware
- Offering a consistent configuration to customers
  - CPU performance
  - Amount of memory
  - Storage
  - Network bandwidth
- Offering additional isolation (reliability)
- **Virtualization of resources happen at different levels based on the service model!**

Virtualization: IaaS

Hardware – lowest level of virtualization

- 1 Physical machine $\Rightarrow$ *n virtual* machines
- Hypervisor: VMWare, VirtualBox, MS HyperV, Xen, etc.
- Run over an OS or "bare-metal"
- Nowadays, virtualization is hardware-assisted: can run at near-native speeds

Virtualized Hardware

- CPU (modern CPUs support virtualization extensions)
- Memory: portions of the RAM of the host machine are reserved
- Storage: virtual hard drives
- Network: virtual network adapters, virtualized networks/subnets
- GPU: for specific applications
- Other devices / pass-thru (e.g., USB)

Virtualization: PaaS / SaaS

Virtualization of the combined resources of a pool of machines (VMs)

- Build over IaaS virtualization layer
- Processing power (CPU)
- Pools of memory
- Distributed data storage
- Virtualized networking and adressing

Elasticity: IaaS

Two approaches to scalability:

- Vertical: more powerful hardware (limited)
- **Horizontal: partitioning / sharding**

Elasticity: IaaS (Infrastructure as a Service)

- Allocating new VM instances
- Deallocating instances which aren't needed anymore
- Allocating storage, RAM, network, etc. (can be properties of the VMs)
- Can be done manually (e.g., through the Amazon EC2 Web Dashboard)...
- ... or automatically at a higher (PaaS) level

Elasticity: PaaS and SaaS

Elasticity: PaaS (Platform as a Service)

- Automatic provisioning of VM/physical resources (IaaS layer) to execute the PaaS application
- The elasticity of the application itself might or might not be done automatically
  - e.g., for a request-based application, the PaaS "execution layer" could provision enough resources to satisfy the amount of requests

Elasticity: SaaS (Software as a Service)

- Fully managed provisioning of the PaaS layer
  - e.g., Gmail will provision enough combined resources at the PaaS layer, which in turn will provision enough resources at the IaaS layer

Data Storage in the Cloud

Data Storage in the Cloud

How data can be stored across different nodes?

- Distributed File Systems
  - Google FS, Hadoop
  - Provides file-system like abstractions in a distributd manner
- Block Storage
  - Amazon S3 (storage of objects, can be files)
- Databases:
  - SQL
  - NoSQL (e.g., Key-value Stores, MongoDB, etc.)

Data Storage in the Cloud: Properties

- Scalability
- High availability
- Low latency
- Durability
- Fault tolerant
- Predictable costs

Data Storage in the Cloud: Properties

- Scalability
- High availability
- Low latency
- Durability
- Fault tolerant
- Predictable costs

Tradeoff: the CAP Theorem

- Consistency
- Availability
- Partition tolerance

Pick only two :-)

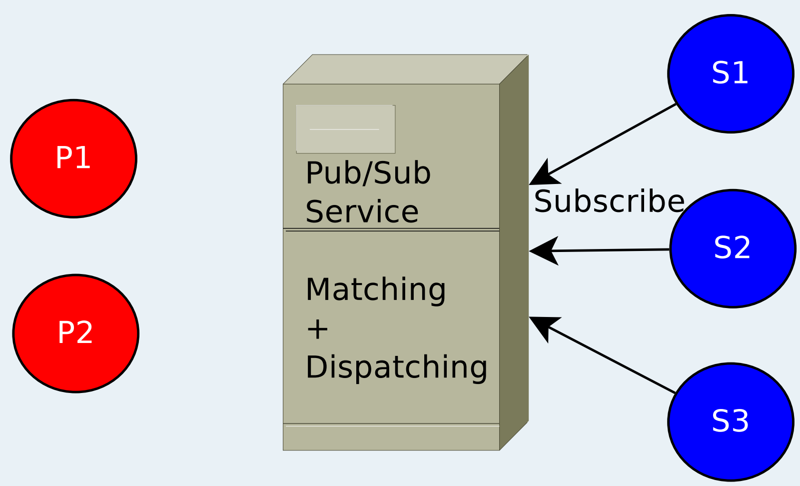
- Cloud storage systems often opt for **eventual consistency**

Communications: Publish/Subscribe

Publish/Subscribe Paradigm

- Provides an elegant way to decouple content producers (publishers) from content consumers (subscribers)
- Publications are matched against subscriptions
- Many *flavours* of publish/subscribe

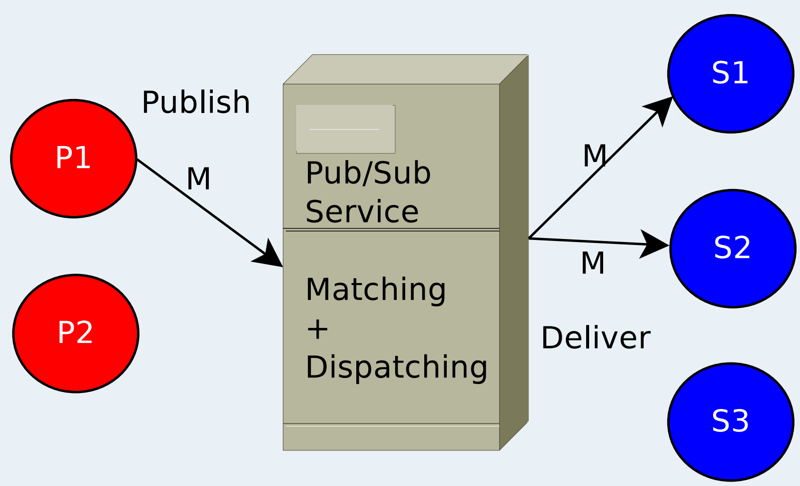
Publish/Subscribe - Example



Publish/Subscribe Paradigm

- Provides an elegant way to decouple content producers (publishers) from content consumers (subscribers)
- Publications are matched against subscriptions
- Many *flavours* of publish/subscribe

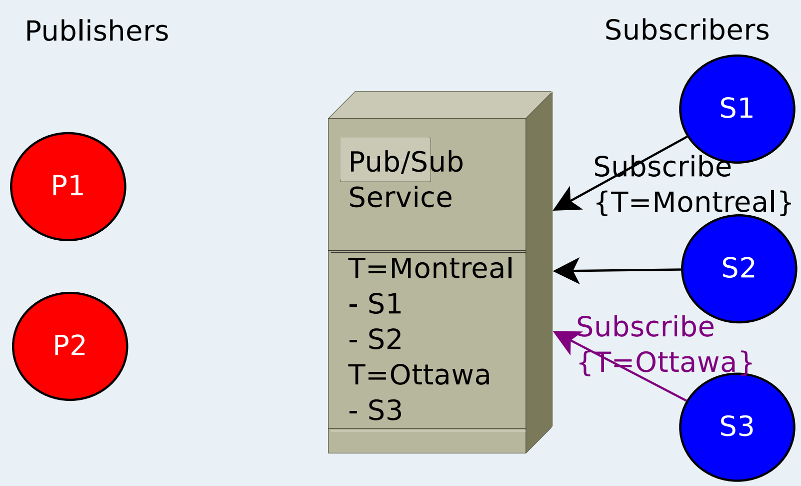
Publish/Subscribe - Example



Topic-Based Publish/Subscribe

- Very common flavour of pub/sub
- Subscription language: a key (topic name)
- Publications tagged with a topic T, sent to all subscribers of T

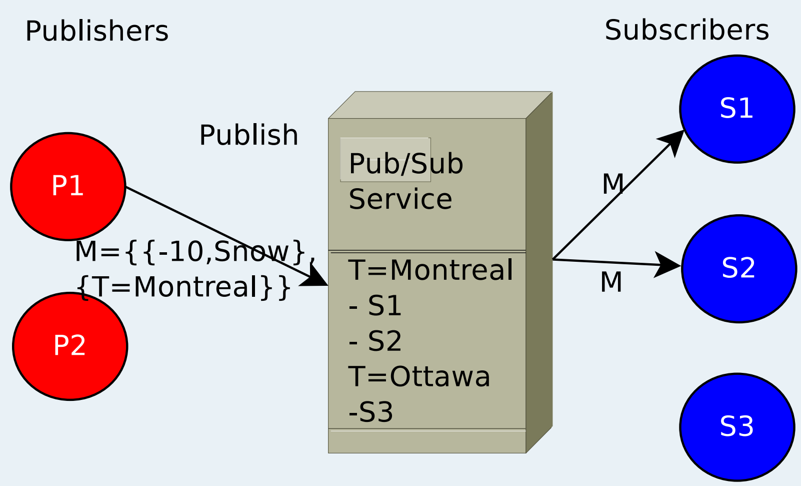
Example - Weather Reports



Topic-Based Publish/Subscribe

- Very common flavour of pub/sub
- Subscription language: a key (topic name)
- Publications tagged with a topic T, sent to all subscribers of T

Example - Weather Reports



Applications of Topic-Based Pub/Sub

Traffic alert systems

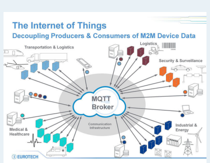
Weather alert systems

Mobile notif. frameworks

Google cloud messaging

Social networks

IoT



Multiplayer Games

Desirable properties:

- Scalability & Elasticity
- Low Latency
- Reduced & Predictable Costs

Dynamoth [ICDCS 2015]: Scalability & Elasticity

Cloud topic-based publish/subscribe service for latency-constrained applications

- Massive Multiplayer Online Games

Dynamoth [ICDCS 2015]: Scalability & Elasticity

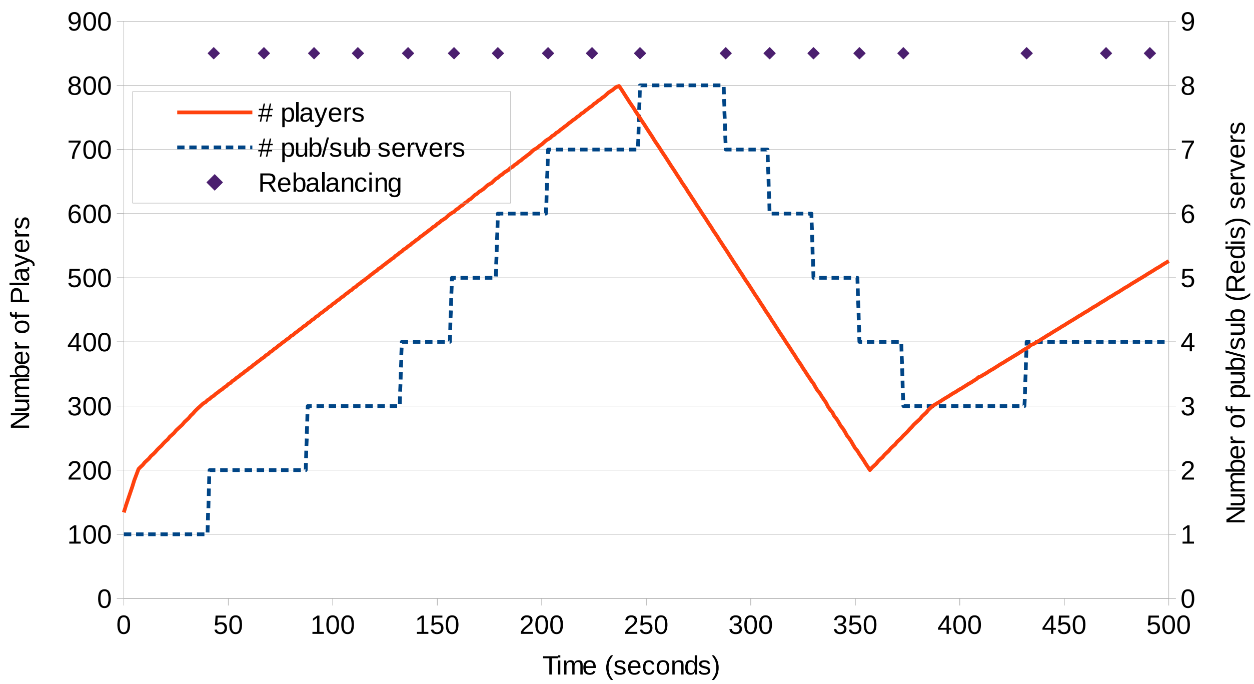
Cloud topic-based publish/subscribe service for latency-constrained applications

- Massive Multiplayer Online Games

Scalability & Elasticity

- Load increases (#players) → more publishers, subscribers and publications
  - Spawn more servers
  - Migrate load accross servers
- Load varies over time
  - Free and add resources

Dynamoth – Results (Elasticity)



MultiPub [ICDCS 2017]: Latency & Cost Optimization

Challenge

- More and more applications are global-scale
- Some applications have very strict latency needs
  - Availability of clouds in several regions
- Cloud usage incurs bandwidth costs
  - Regions have highly varying bandwidth costs

EC2 Cloud Regions (~2016)

Region	Location	$ per outgoing GB
us-east-1	N. Virginia	0.09
us-west-1	N. California	0.09
us-west-2	Oregon	0.09
eu-west-1	Ireland	0.09
eu-central-1	Frankfurt	0.09
ap-northeast-1	Tokyo	0.14
ap-northeast-2	Seoul	0.126
ap-southeast-1	Singapore	0.12
ap-southeast-2	Sydney	0.14
sa-east-1	Sao Paulo	0.25

Batch Processing: Map/Reduce

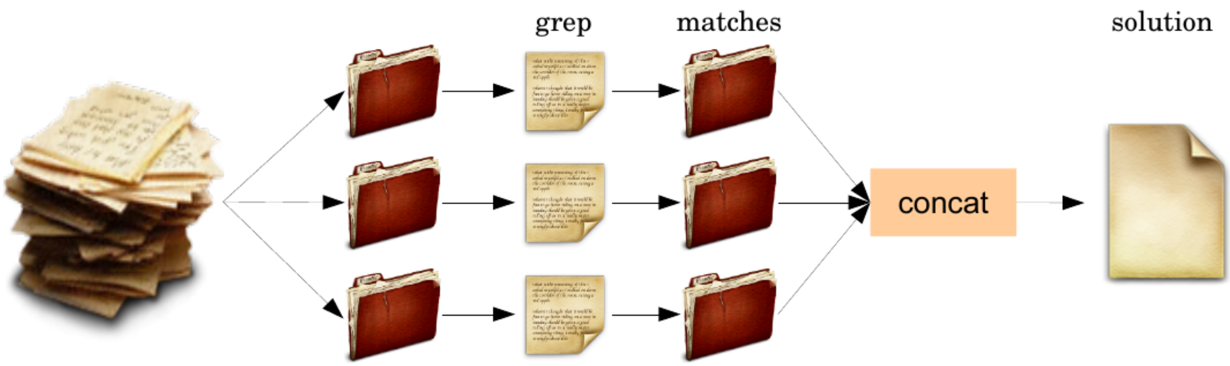
Map/Reduce (Source)

- Functional Decomposition:
  - Breaking a large problem broken into a set of small problems
- Each small problem:
  - can be solved by a functional transformation of input data
  - can be executed in complete isolation (parallel computing)

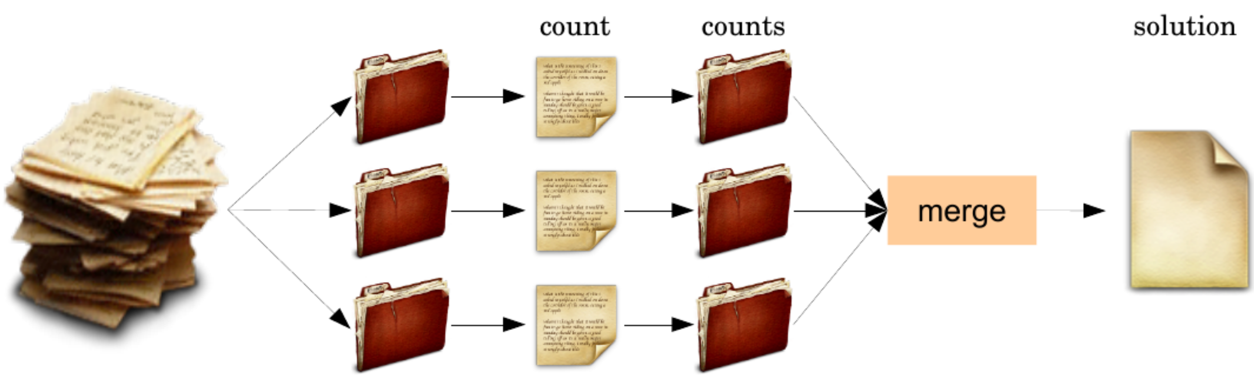
Examples (next slides) – what do these Linux programs do?

- **grep**
- **wc (word count)**

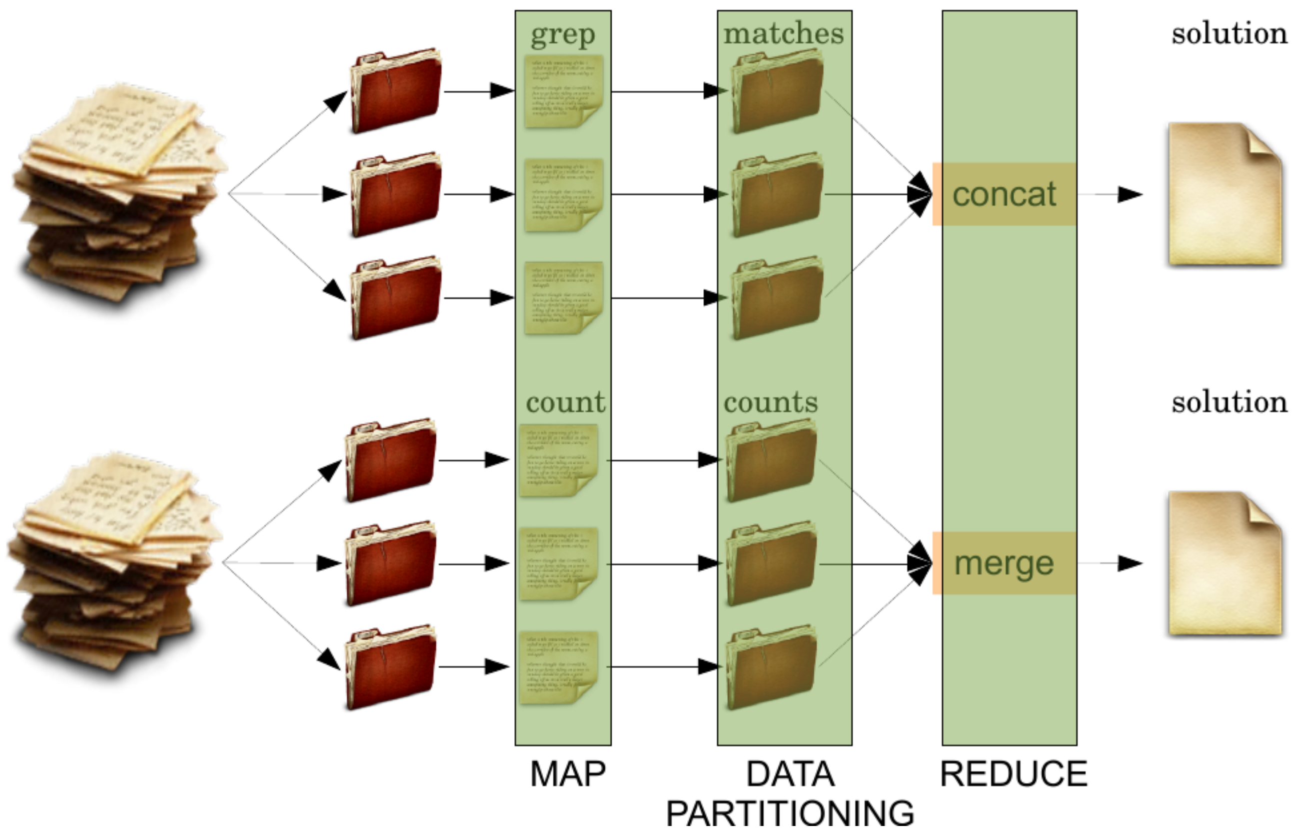
grep with MapReduce



wc with MapReduce



grep and wc with MapReduce



Serveless Computing / Function as a Service

Function as a Service (FaaS)

- Application made of a set of functions
- Executed upon certain events being triggered
  - Web request
  - File upload
  - Change to DB
  - Timer
- Executed within containers (thin VMs)
  - Full isolation
  - FaaS functions are **stateless!**
    - Changes in state must be persisted to durable storage
- Example: Amazon Lambda, Google Cloud Functions, MS Azure Functions

Class Activity :-)

Class Activity – Load Balancing HTTP Requests

- Consider a cloud with 4 machines that accepts HTTP requests. Each requests triggers some "heavy" processing, and one server would not be enough to satisfy the demand – hence, we need to spread the requests between the 4 node servers.
- We will not be using a real could – you will be emulating this setup on your local machine.

Class Activity – Load Balancing HTTP Requests (2)

- wasteCycles.js is a Node program that serves the following HTTP requests:
  - /cycles: occupies the CPU for one second
  - (any string ending with .html or .js): returns the contents of the file (as you did in class)
  - Due to the single-threadness of Node, if you execute more than one request per second, the program will delay the processing of further requests
  - The program takes as input a port number to listen to.
- waste.js is a Node program that submits a request to *wasteCycles.js* every 800ms.
- Upon running waste.js, you will notice that the processing time goes well above 1000ms

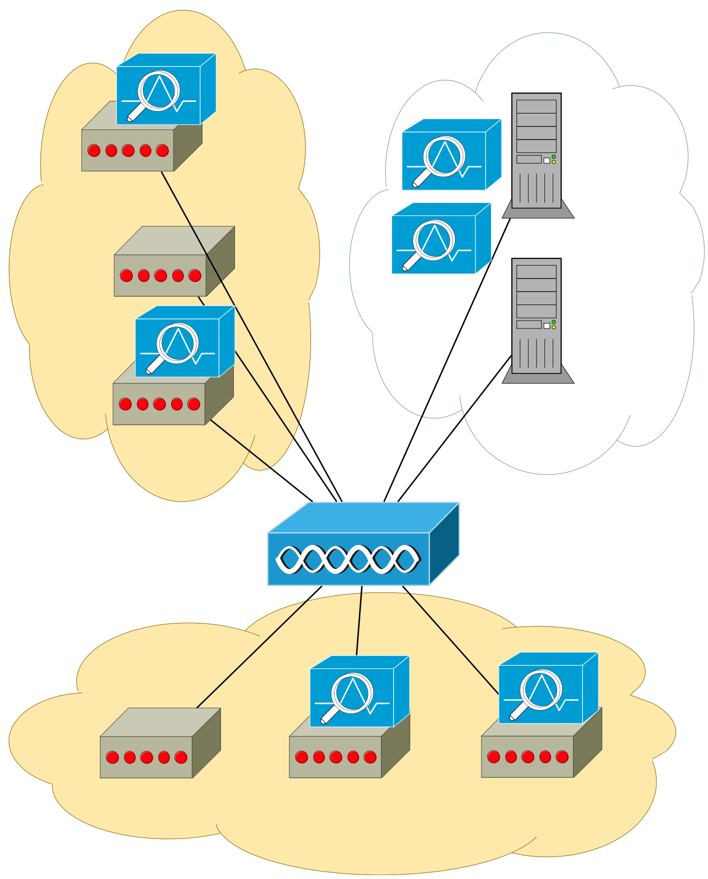
Your task:

- You should launch four instances of wasteCycles.js to emulate four cloud servers
  - node wasteCycles.js 8080, node wasteCycles.js 8081, node wasteCycles.js 8082, node wasteCycles.js 8083
- Implement loadBalancer.js , a Node program accepts requests on port 8079 and submits them in a round-robin fashion to one of the node servers (wasteCycles)
- To test, modify the port number in waste.js to be 8079
- If your implementation is correct, the processing time should be stable around 1000ms

Edge Computing

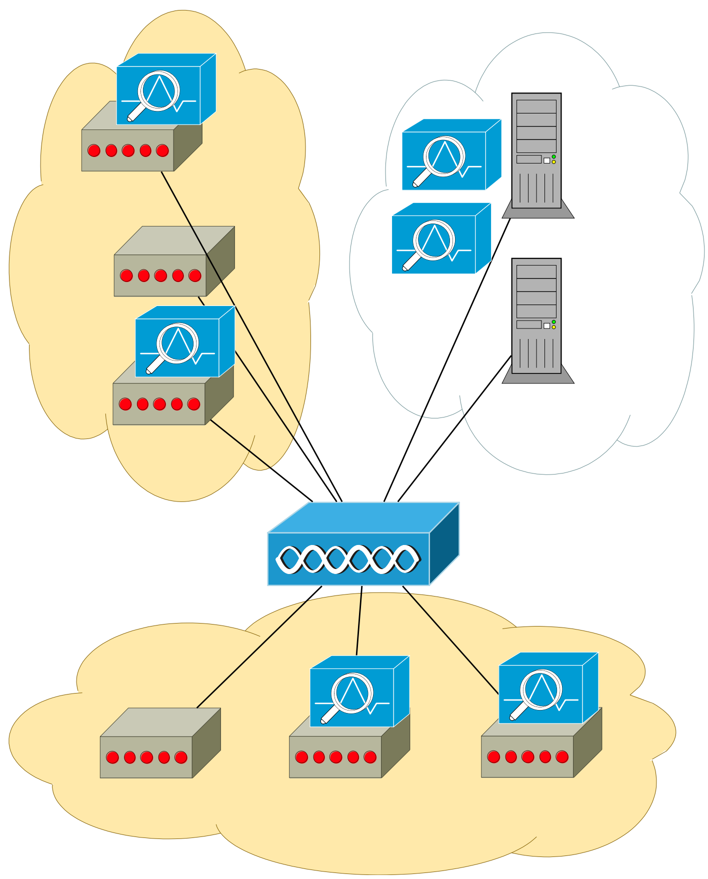
Edge Computing

- IoT devices are getting more and more powerful!



Edge Computing

- IoT devices are getting more and more powerful!
- Running computations *on the devices* (*edge computing*)

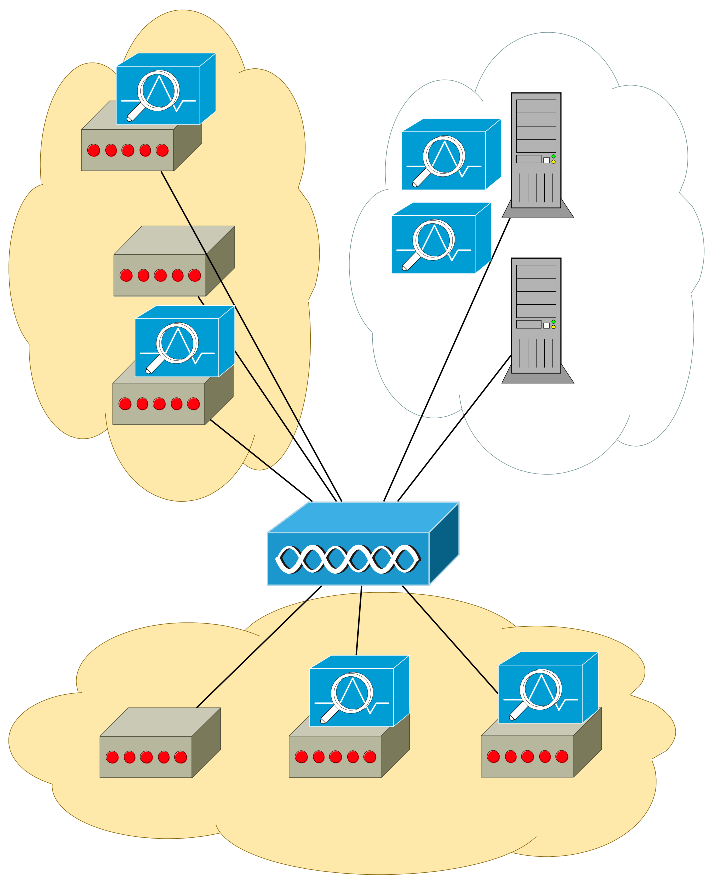


Edge Computing

- IoT devices are getting more and more powerful!
- Running computations *on the devices* (*edge computing*)

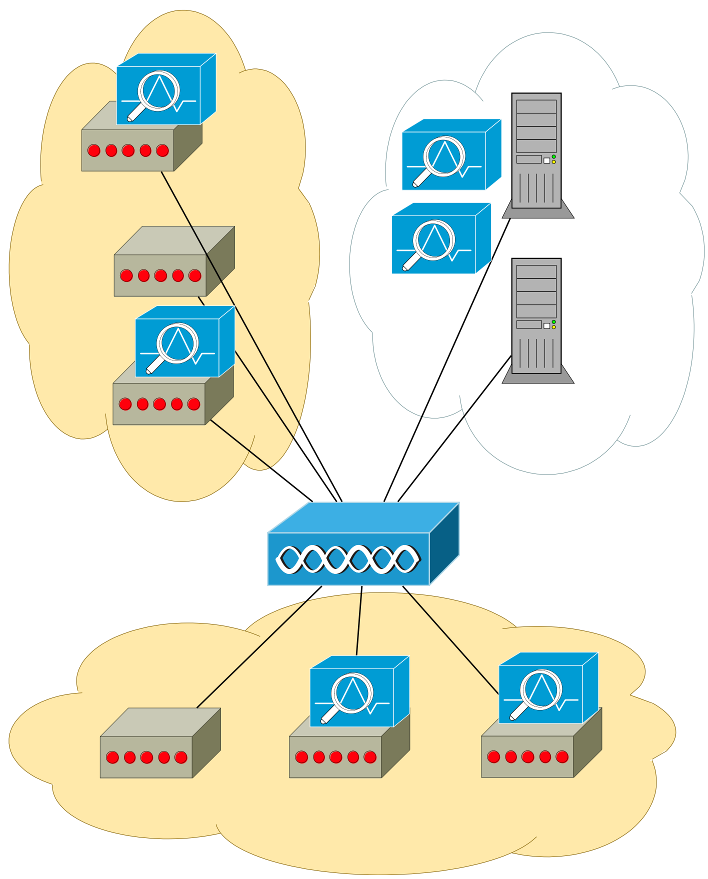
Cloud: Challenges and Limitations

- Connectivity
  - Avoid reliance towards the cloud



Edge Computing

- IoT devices are getting more and more powerful!
- Running computations *on the devices* (*edge computing*)

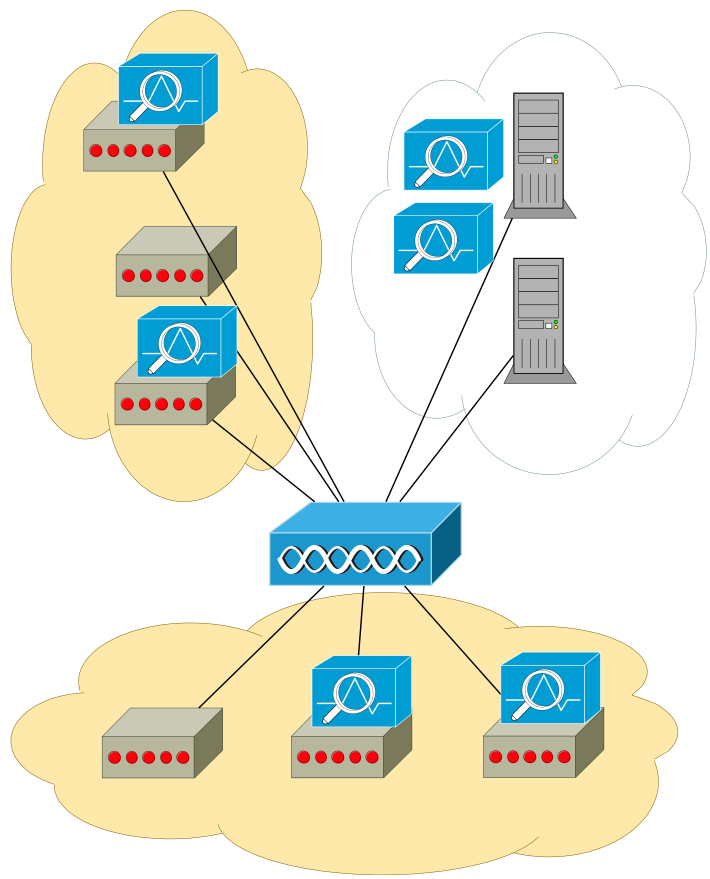


Cloud: Challenges and Limitations

- Connectivity
  - Avoid reliance towards the cloud
- Cost effectiveness
  - Saving costs of using the cloud

Edge Computing

- IoT devices are getting more and more powerful!
- Running computations *on the devices* (*edge computing*)



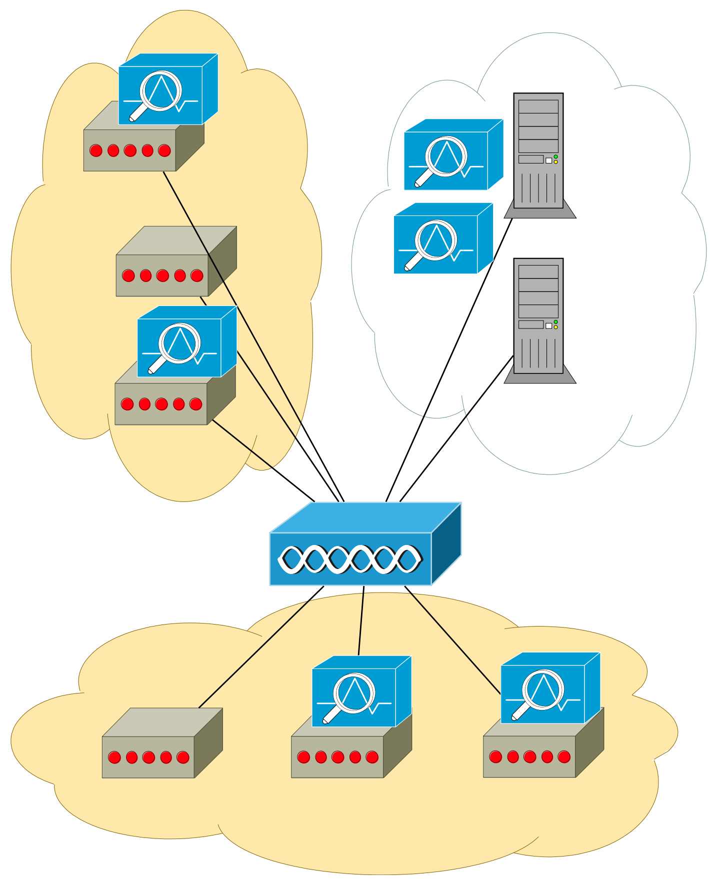
Cloud: Challenges and Limitations

- Connectivity
  - Avoid reliance towards the cloud
- Cost effectiveness
  - Saving costs of using the cloud
- Performance
  - Reduced latencies

ThingsJS: an IoT Runtime Middleware

- Current state of the IoT:
  - Many devices
  - Different APIs, frameworks, languages
  - Incompatible protocols

$\Downarrow$

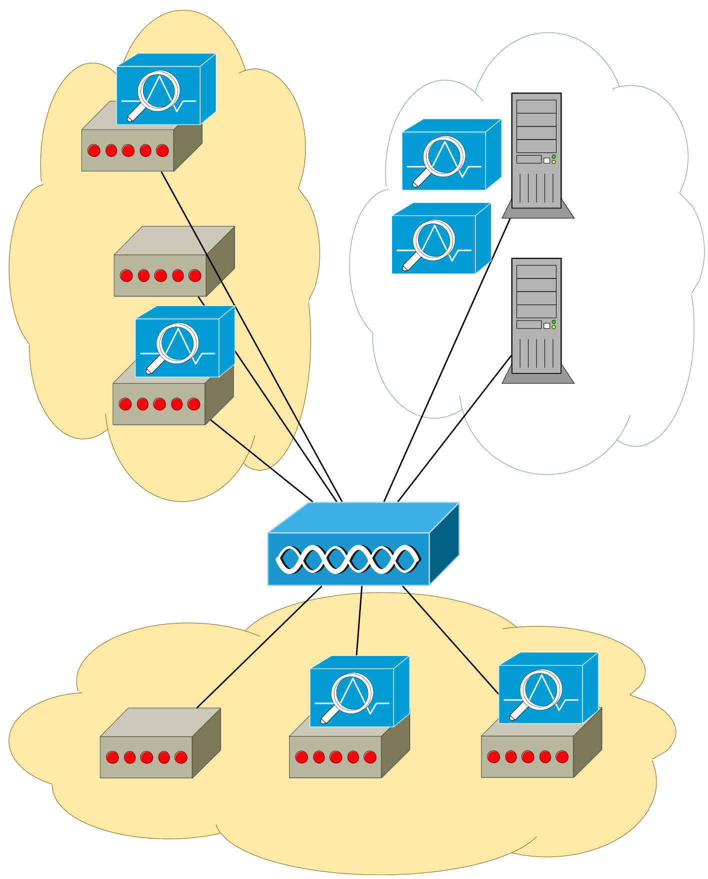


ThingsJS: an IoT Runtime Middleware

- Current state of the IoT:
  - Many devices
  - Different APIs, frameworks, languages
  - Incompatible protocols

$\Downarrow$

- **ThingsJS**: a framework for developing and deploying *high-level* applications on IoT devices (edge computing)



ThingsJS: an IoT Runtime Middleware

- Current state of the IoT:
  - Many devices
  - Different APIs, frameworks, languages
  - Incompatible protocols

$\Downarrow$

- **ThingsJS**: a framework for developing and deploying *high-level* applications on IoT devices (edge computing)

JavaScript

- Programmers are typically more productive in higher-level languages
- JavaScript: strong user base

ThingsJS: an IoT Runtime Middleware

- Current state of the IoT:
  - Many devices
  - Different APIs, frameworks, languages
  - Incompatible protocols

$\Downarrow$

- **ThingsJS**: a framework for developing and deploying *high-level* applications on IoT devices (edge computing)

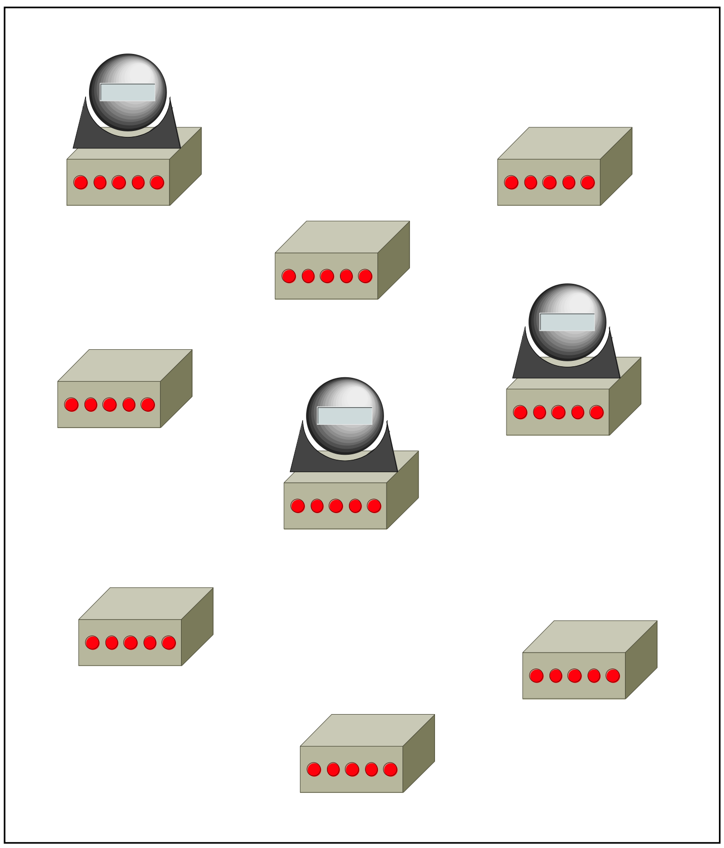
JavaScript

- Programmers are typically more productive in higher-level languages
- JavaScript: strong user base

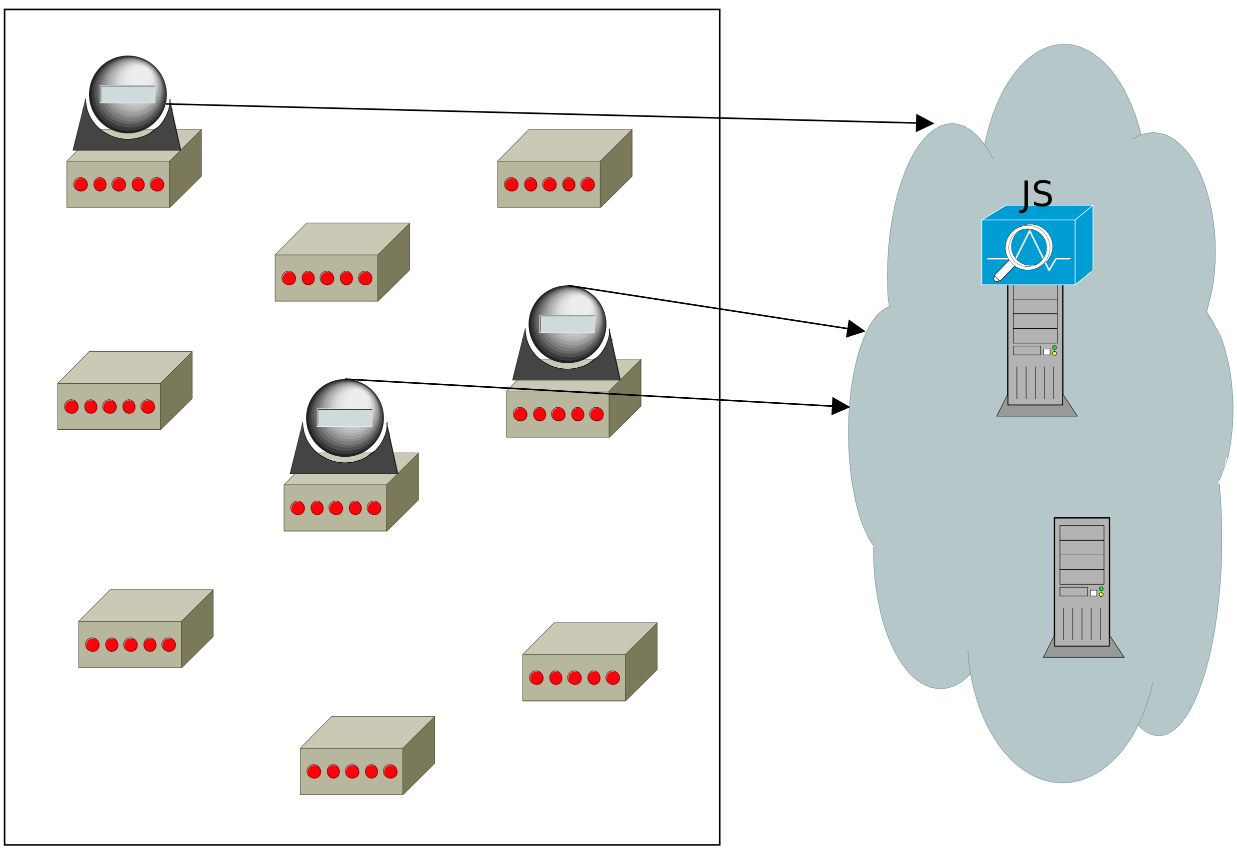
JavaScript VMs on IoT

- Samsung IoT.js, Intel XDK, DukServer, Smart.js
- Node.js, ChakraCore, SpiderMonkey on IoT devices

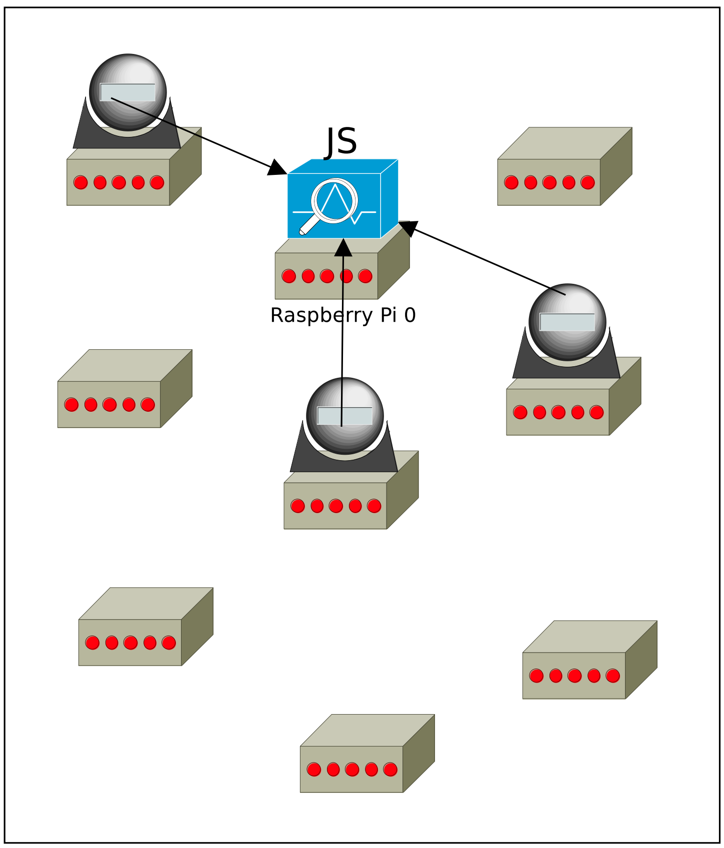
Motivational Use Case: Video-surveillance Application



Motivational Use Case: Video-surveillance Application



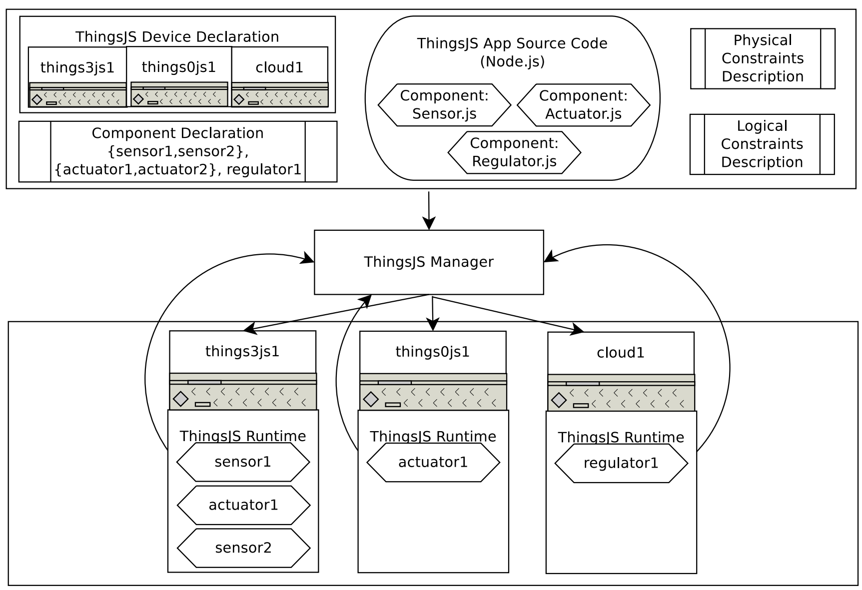
Motivational Use Case: Video-surveillance Application



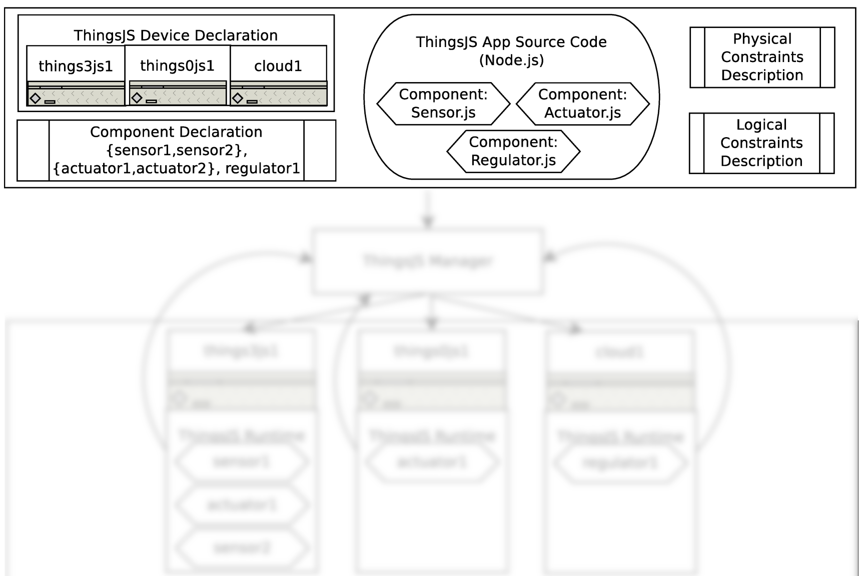
ThingsJS:

Executing High-Level Applications on IoT/Edge devices.

Architectural Model

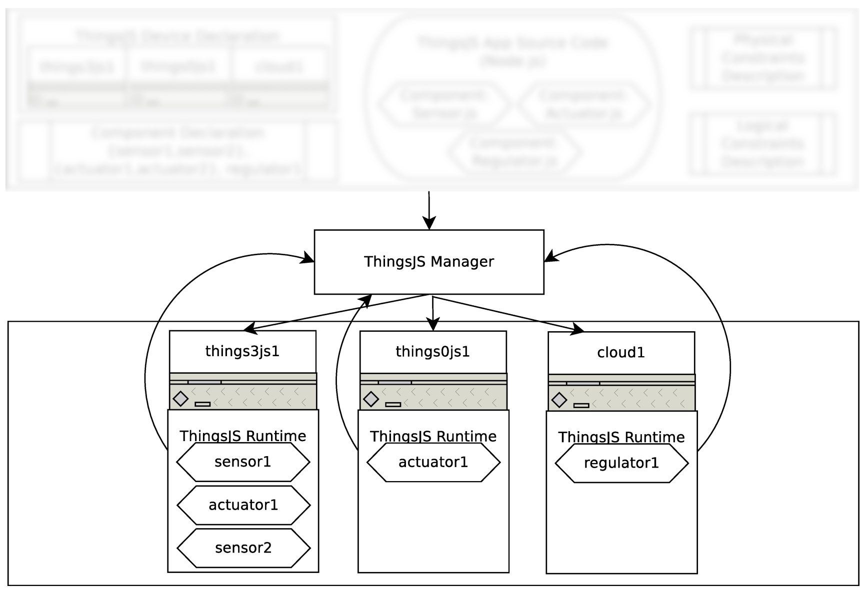


Architectural Model



- **ThingsJS Application**: package containing components, metadata, constraints (more later)

Architectural Model



- **ThingsJS Application**: package containing components, metadata, constraints (more later)
- **ThingsJS Infastructure**: APIs, services, scheduling, etc. (more later)

JavaScript-Centric Programming Model

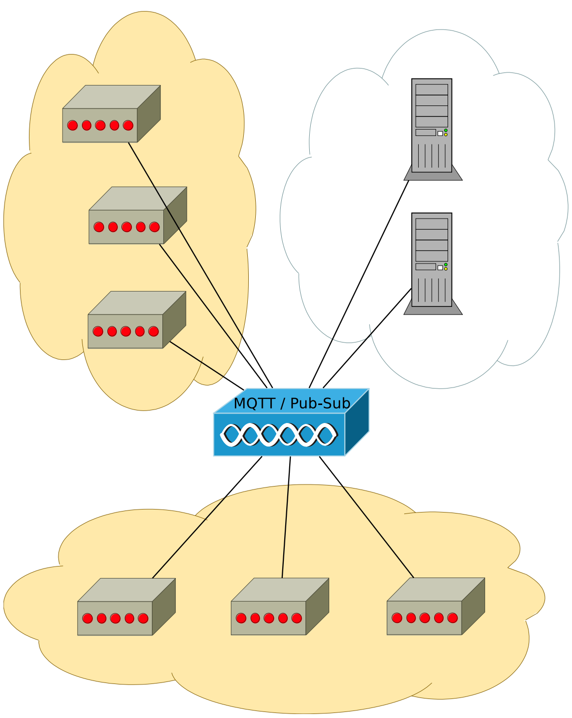

```
/*things.meta
requiredMemory: 50
*/

/* Basic ThingsJS-compatible factorial calculator.
 * Note we cannot use a while loop or a recursive function as doing so would
 * block the thread, preventing the migration signal from being processed.
 * We use a setImmediate to call the next step of the factorial computation.
 *
 * Set the "target" variable to specify the argument.
 * In this example this code will compute factorial(100000).
 * */
var target = 100000;
var timer;
var count = 0;
var digits = [ 1 ];

function factorial(){
    count ++;
    var carry = 0;
    var product = 0;
    for (var i=0; i < digits.length; i++){
        product = digits[i] * count;
        product += carry;
        digits[i] = product % 10;
        carry = Math.floor(product / 10);
    }
    while (carry > 0){
        digits.push(carry % 10);
```

Global File-System

- Unified file system (built over MongoDB)
- Usable by all IoT/edge apps
- Stores text and binary files in a folder hierarchy:
  - Source code for application
  - Configuration files
  - Live data (similar to Unix)
- Also stores JavaScript (JSON) objects

Pub/Sub (MQTT) Communication Service

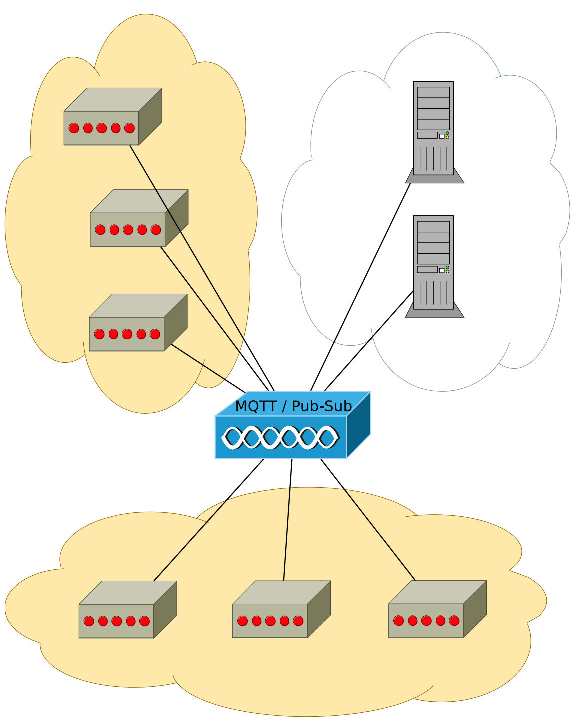
Centralized Pub/Sub



$\Rightarrow$

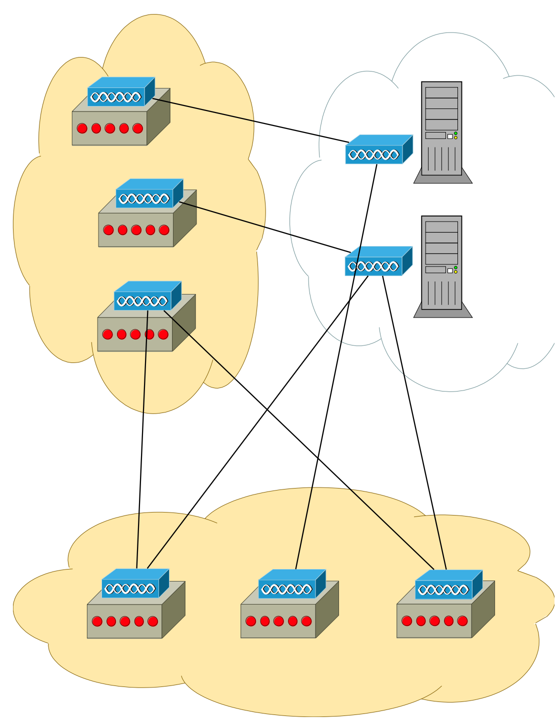
Pub/Sub (MQTT) Communication Service

Centralized Pub/Sub



$\Rightarrow$

Distributed Pub/Sub



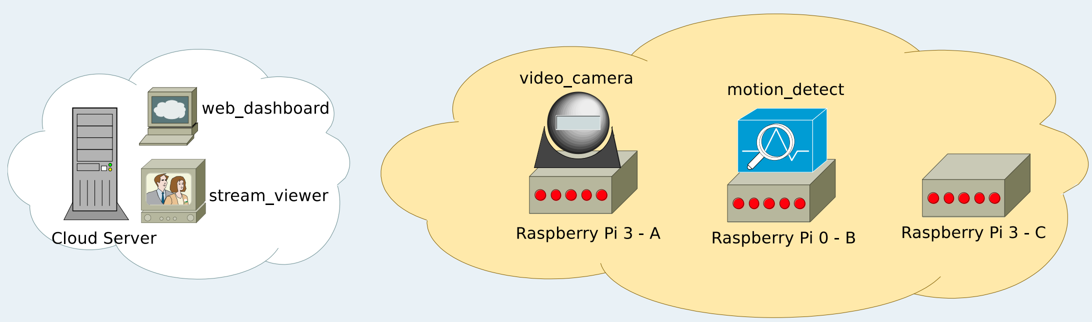
Scheduling IoT Applications on Edge & Cloud Devices

1. Given a set of IoT applications (“components”)
2. Given a set of devices
3. Given a set of constraints

Scheduling IoT Applications on Edge & Cloud Devices

1. Given a set of IoT applications ("components")
2. Given a set of devices
3. Given a set of constraints

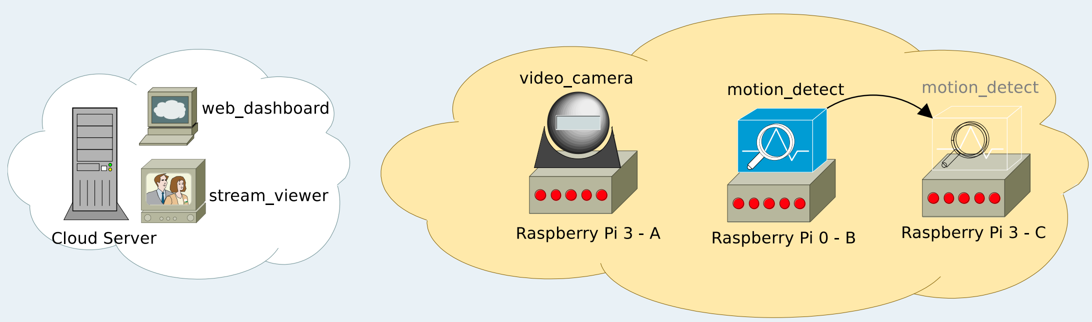
What is the optimal mapping of components to devices?



Scheduling IoT Applications on Edge & Cloud Devices

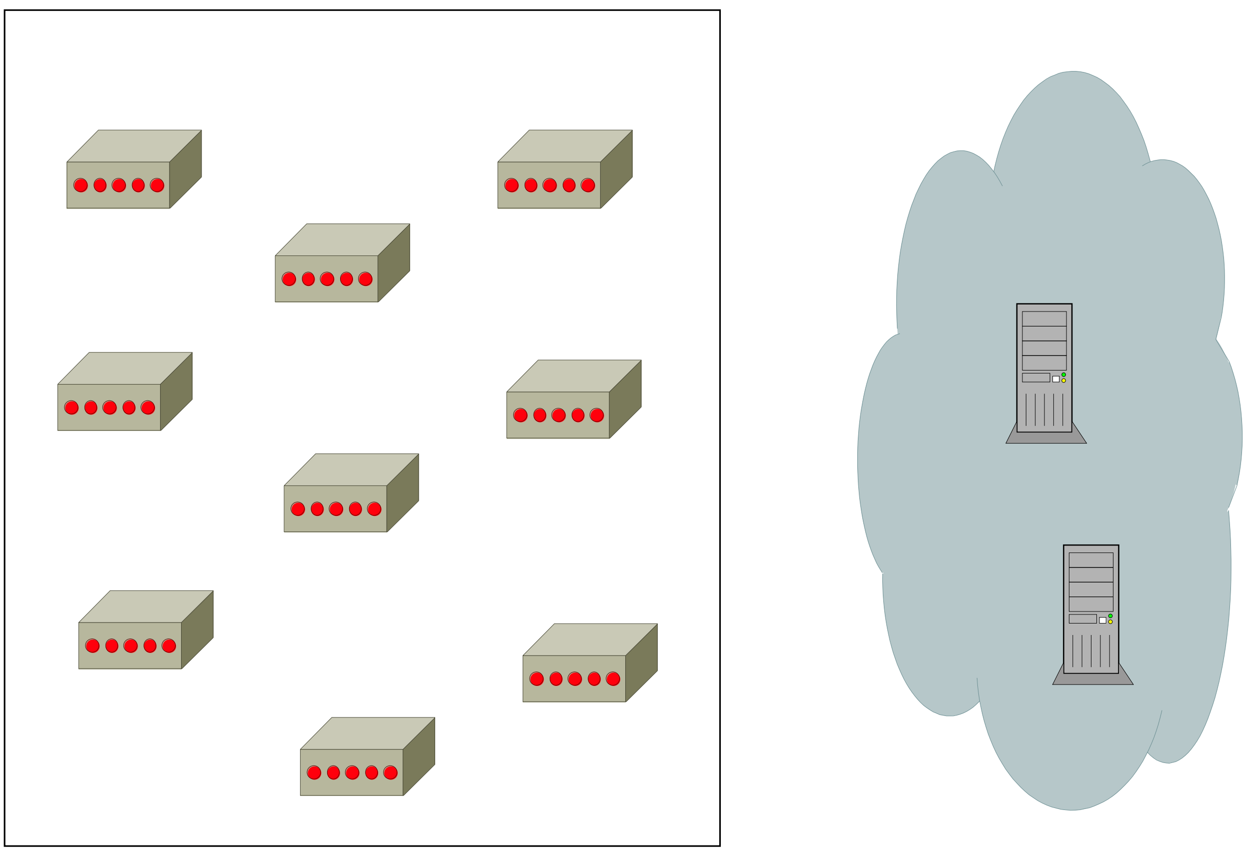
1. Given a set of IoT applications ("components")
2. Given a set of devices
3. Given a set of constraints

What is the optimal mapping of components to devices?

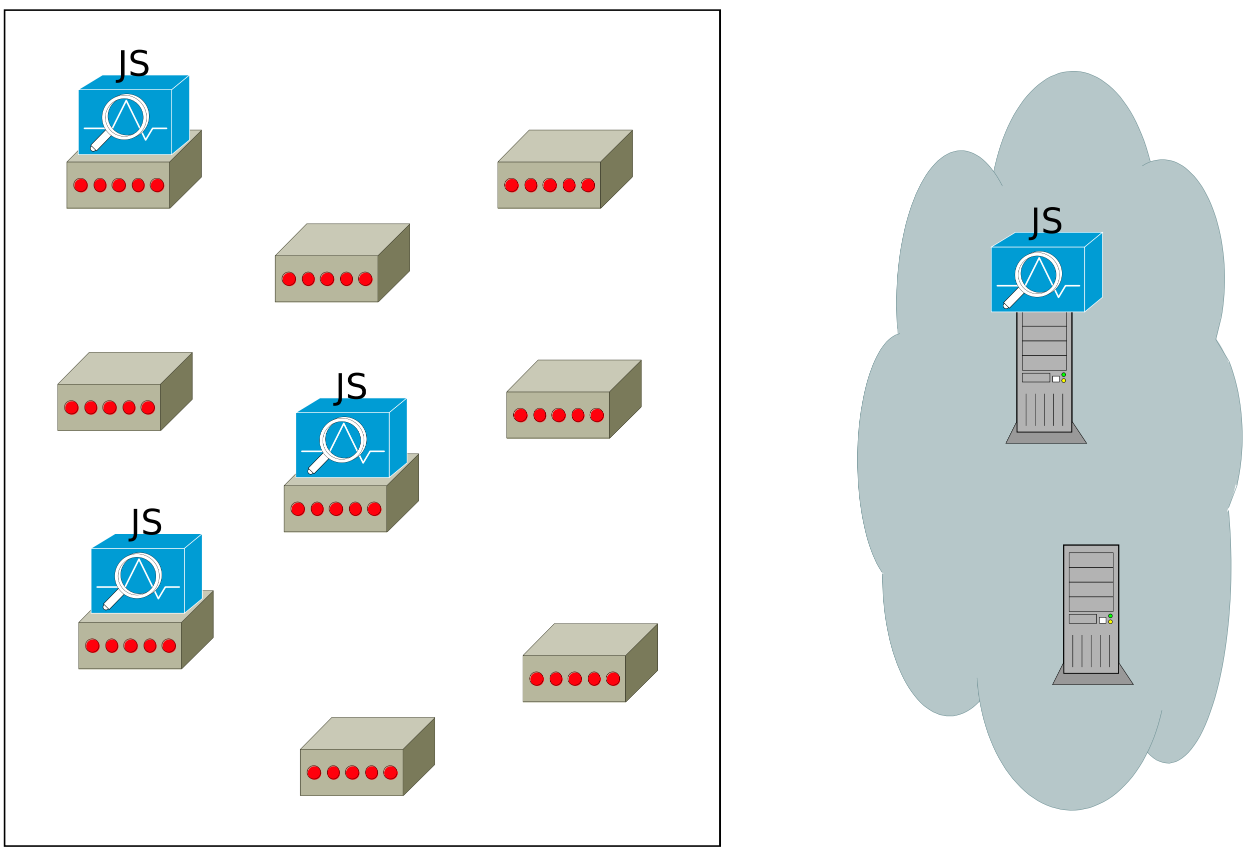


⇒ *What if we must dynamically alter the current schedule?*

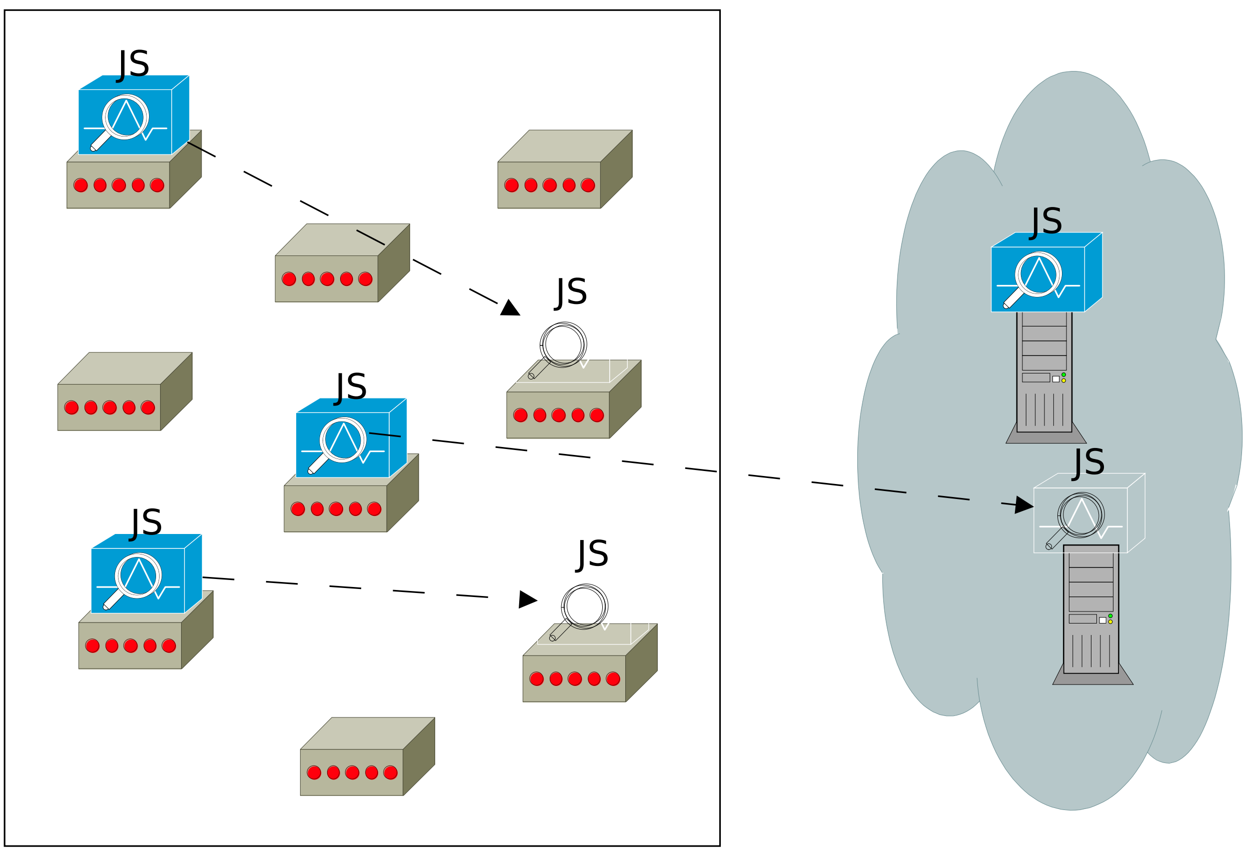
Migrating JavaScript IoT Applications: ThingsMigrate



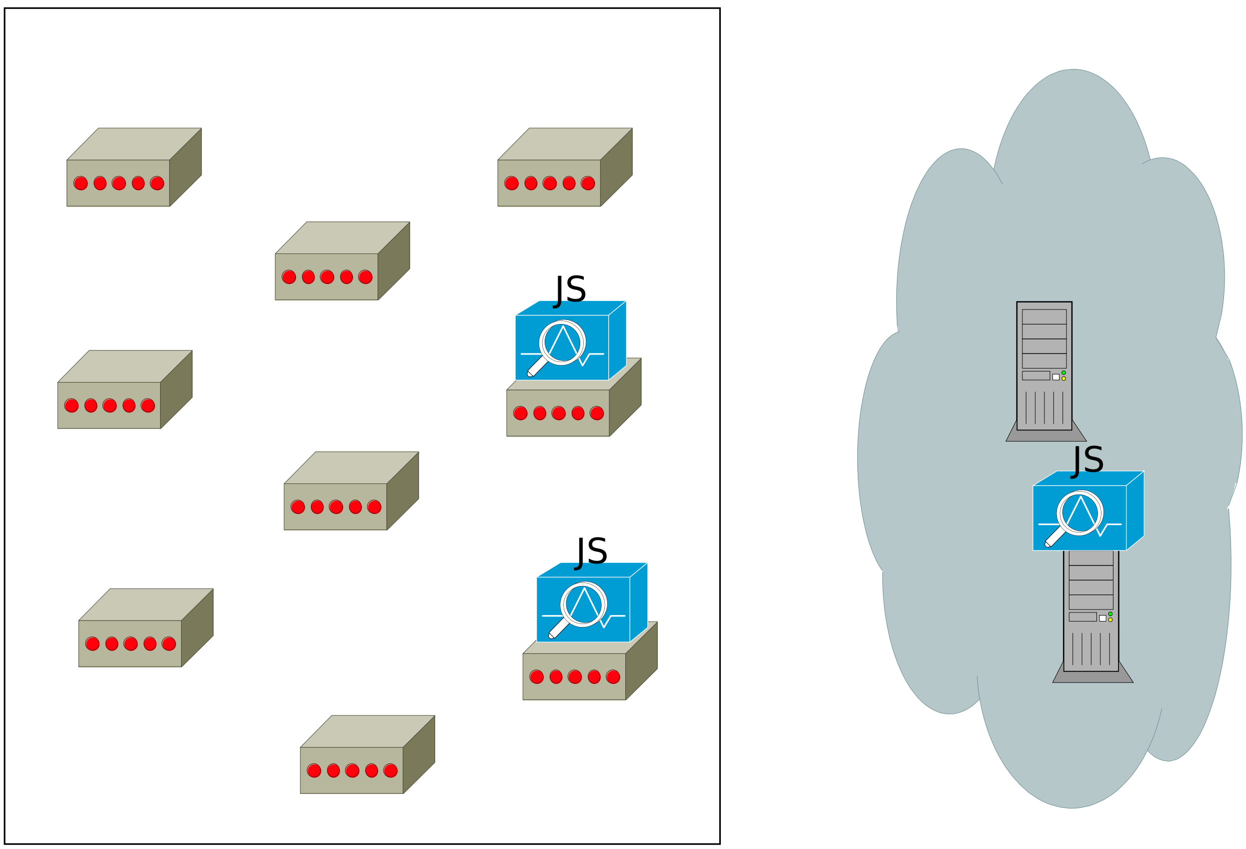
Migrating JavaScript IoT Applications: ThingsMigrate



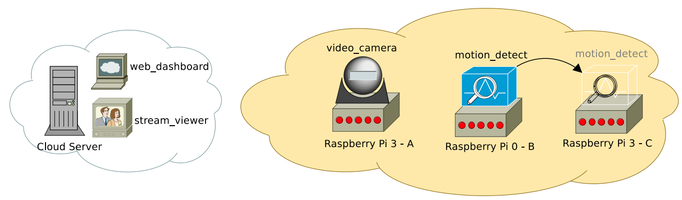
Migrating JavaScript IoT Applications: ThingsMigrate



Migrating JavaScript IoT Applications: ThingsMigrate

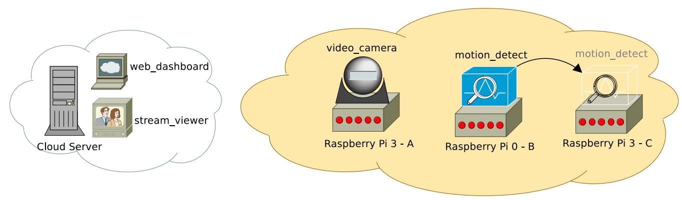


ThingsMigrate: Motivation & Requirements



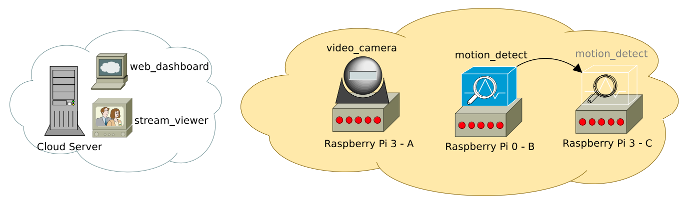
- Supporting migration on resource-constrained IoT devices

ThingsMigrate: Motivation & Requirements



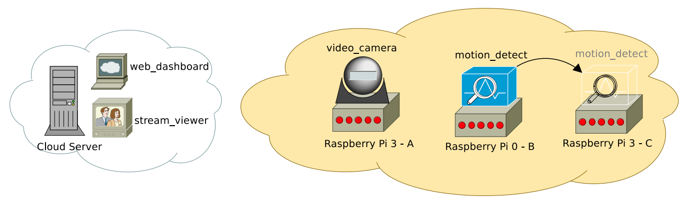
- Supporting migration on resource-constrained IoT devices
- More flexible than terminating and restarting

ThingsMigrate: Motivation & Requirements



- Supporting migration on resource-constrained IoT devices
- More flexible than terminating and restarting
- Migration between devices, and between devices and the cloud

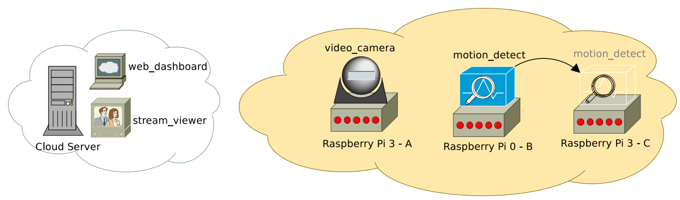
ThingsMigrate: Motivation & Requirements



- Supporting migration on resource-constrained IoT devices
- More flexible than terminating and restarting
- Migration between devices, and between devices and the cloud

- Portability: heterogeneous devices, cloud (*cloud-edge* computing)
  - No modifications to VM

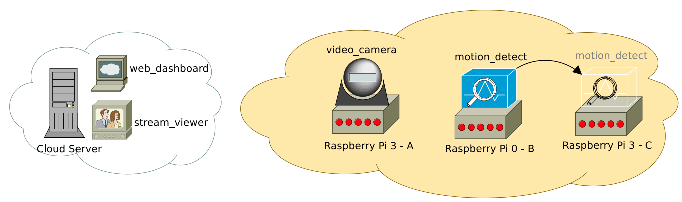
ThingsMigrate: Motivation & Requirements



- Supporting migration on resource-constrained IoT devices
- More flexible than terminating and restarting
- Migration between devices, and between devices and the cloud

- Portability: heterogeneous devices, cloud (*cloud-edge* computing)
    - No modifications to VM
- Supporting *stateful* applications

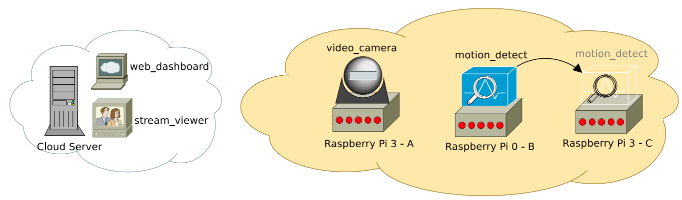
ThingsMigrate: Motivation & Requirements



- Supporting migration on resource-constrained IoT devices
- More flexible than terminating and restarting
- Migration between devices, and between devices and the cloud

- Portability: heterogeneous devices, cloud (*cloud-edge* computing)
  - No modifications to VM
- Supporting *stateful* applications
- Publish/Subscribe (MQTT) for all communications

ThingsMigrate: Motivation & Requirements



- Supporting migration on resource-constrained IoT devices
- More flexible than terminating and restarting
- Migration between devices, and between devices and the cloud

- Portability: heterogeneous devices, cloud (*cloud-edge* computing)
  - No modifications to VM
- Supporting *stateful* applications
- Publish/Subscribe (MQTT) for all communications
- JavaScript programs are single-threaded: developers should avoid blocking the main thread

Challenges

Wide heterogeneity of devices, OS and JavaScript VMs!

Challenges

Wide heterogeneity of devices, OS and JavaScript VMs!

Challenge: capturing the state of the JavaScript application

1. Closures / data encapsulation in functions

```
function Counter() {
    var value = 0;

    return function() {
        value += 1;
        return value;
    }
};
var c = Counter();  // value in c is 0
console.log( c() ); // prints 1
console.log( c() ); // prints 2
```

Challenges

Wide heterogeneity of devices, OS and JavaScript VMs!

Challenge: capturing the state of the JavaScript application

1. Closures / data encapsulation in functions
2. Timers

```
function Counter() {
   var value = 0;

   return function() {
      value += 1;
      return value;
   }
};
var c = Counter();  // value in c is 0
console.log( c() ); // prints 1
console.log( c() ); // prints 2
setInterval(function() { c(); },1000);
```

Challenges

Wide heterogeneity of devices, OS and JavaScript VMs!

Challenge: capturing the state of the JavaScript application

1. Closures / data encapsulation in functions
2. Timers
3. Classes and prototypes

```
function Counter() {
   var value = 0;

   return function() {
      value += 1;
      return value;
   }
};
var c = Counter();  // value in c is 0
console.log( c() ); // prints 1
console.log( c() ); // prints 2
setInterval(function() { c(); },1000);
```

Challenges

Wide heterogeneity of devices, OS and JavaScript VMs!

Challenge: capturing the state of the JavaScript application

1. Closures / data encapsulation in functions
2. Timers
3. Classes and prototypes
4. Asynchronous Model (Event-Based)

```
function Counter() {
   var value = 0;

   return function() {
      value += 1;
      return value;
   }
};
var c = Counter();  // value in c is 0
console.log( c() ); // prints 1
console.log( c() ); // prints 2
setInterval(function() { c(); },1000);
```

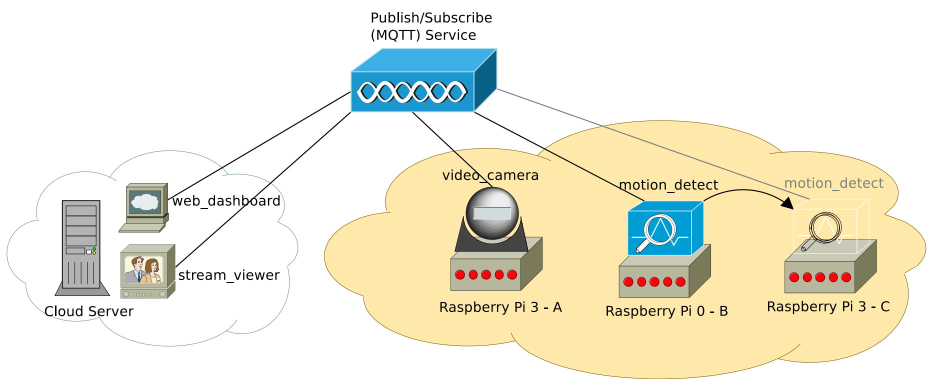
Approach: Code Instrumentation & Reconstruction


```
function Counter() {
    var value = 0;

    return function() {
        value += 1;
        return value;
    }
};
var c = Counter();  // value in c is 0
console.log( c() ); // prints 1
console.log( c() ); // prints 2
setInterval(function() { c(); },1000);
```

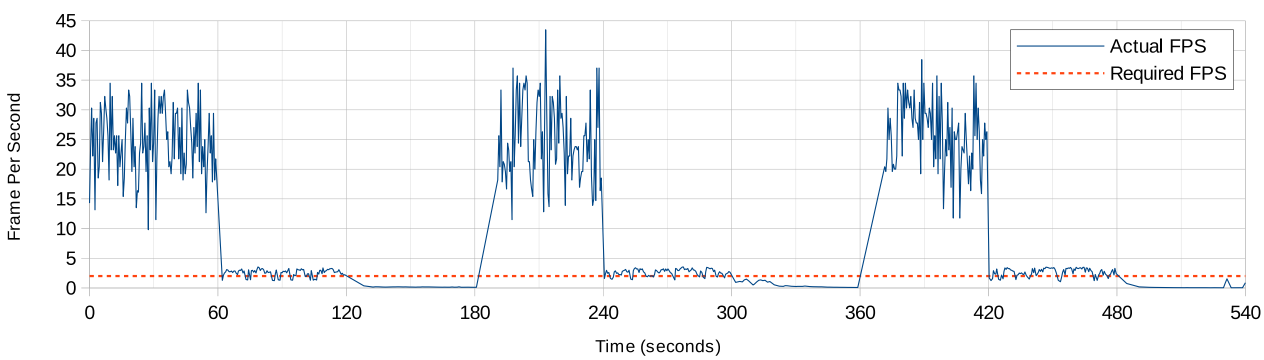
⇓

```
var global = new Scope("global");
function Counter() {
    counter. = new Scope(global, "Counter");
    var value = 0;
    counter.addVar("value", value);

    var anon1 = function() {
        anon1 = new Scope(createcounters, "anon1");
        value += 1;
        anon1.setVar("value", value);

        return value;
    }

    counter.addFunction("anon1", anon1);
    return anon1;
};
```

Case Study: Motion Detection

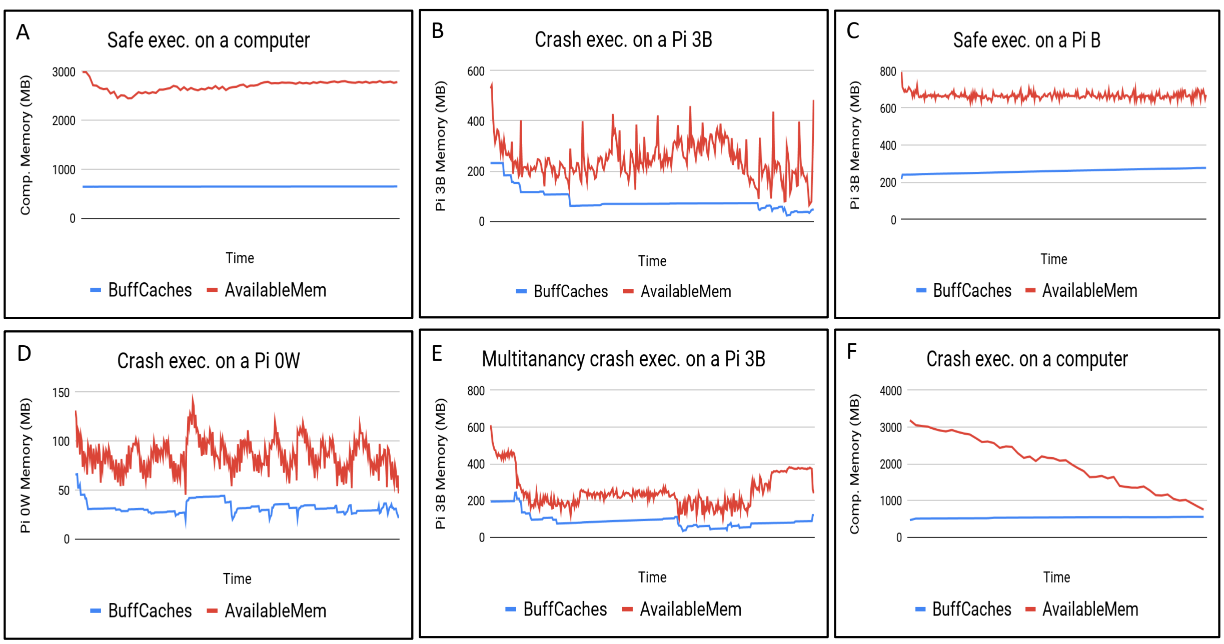


Reachable vs target "frames per second" (FPS):

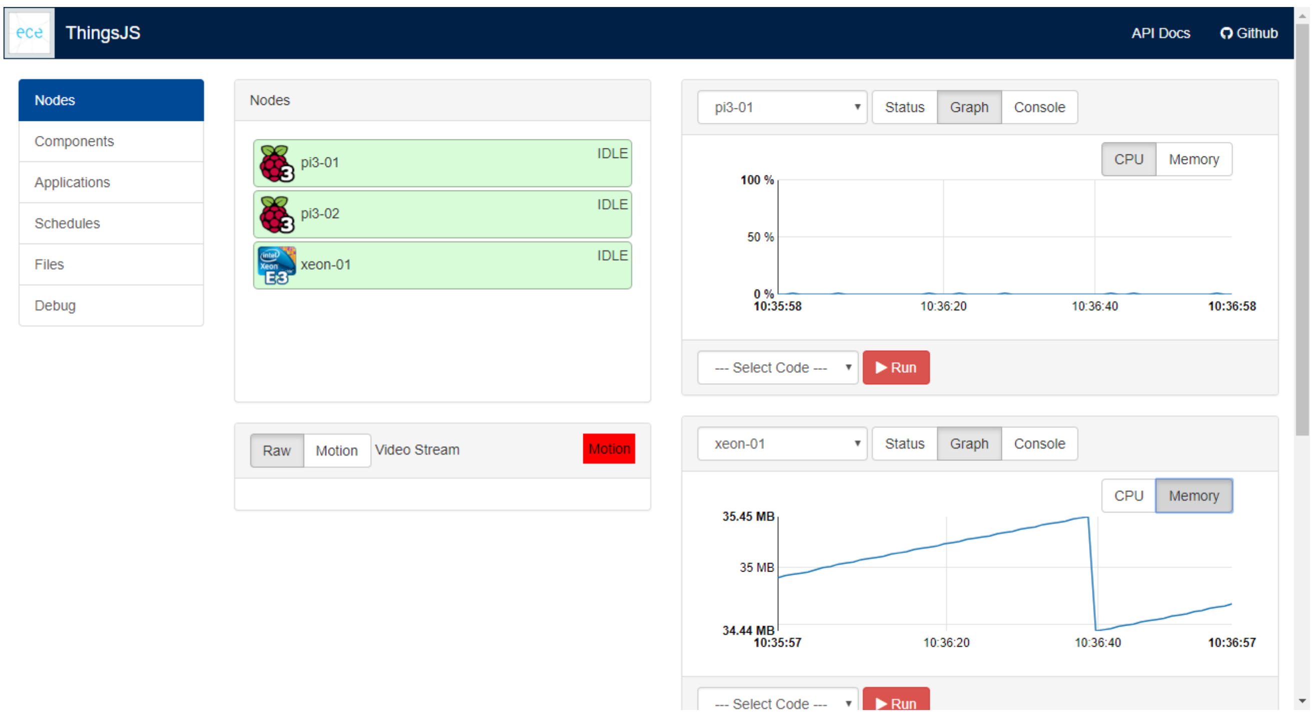


Predicting IoT Application Failures

Predicting failures in high-level apps running on IoT devices

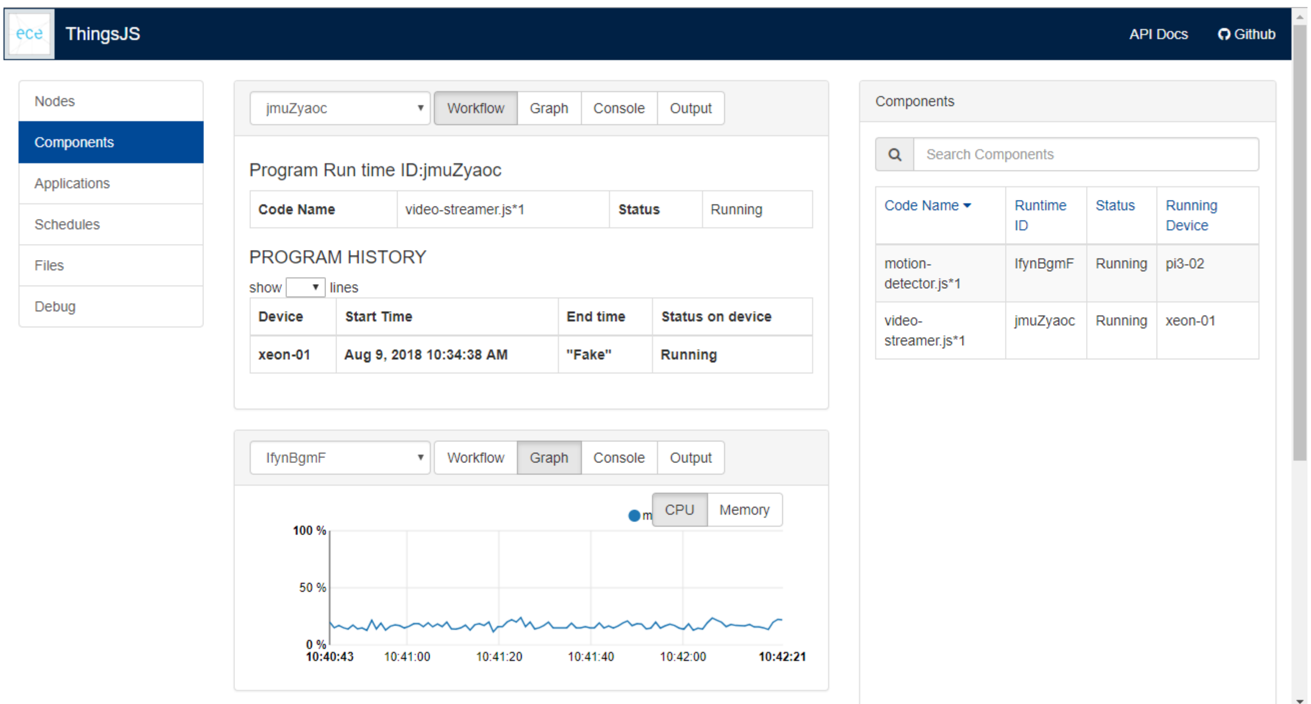


Web Dashboard



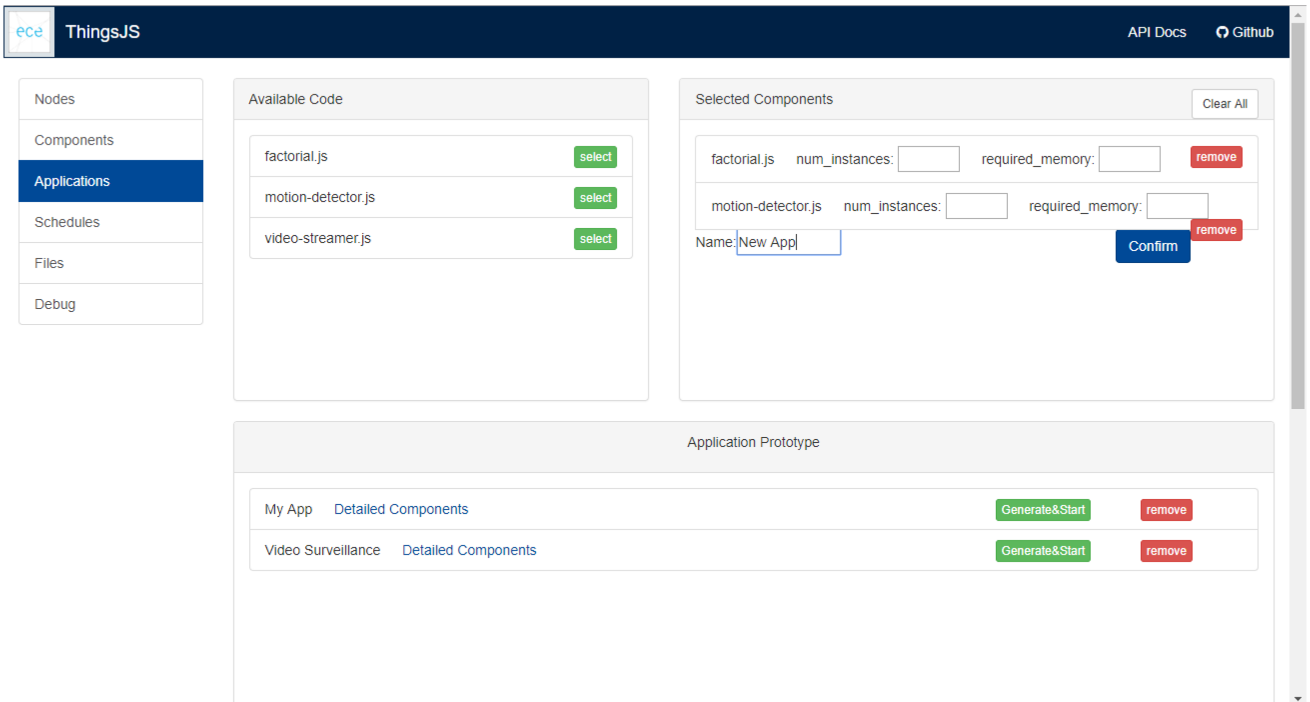
IoT (and cloud) devices

Web Dashboard



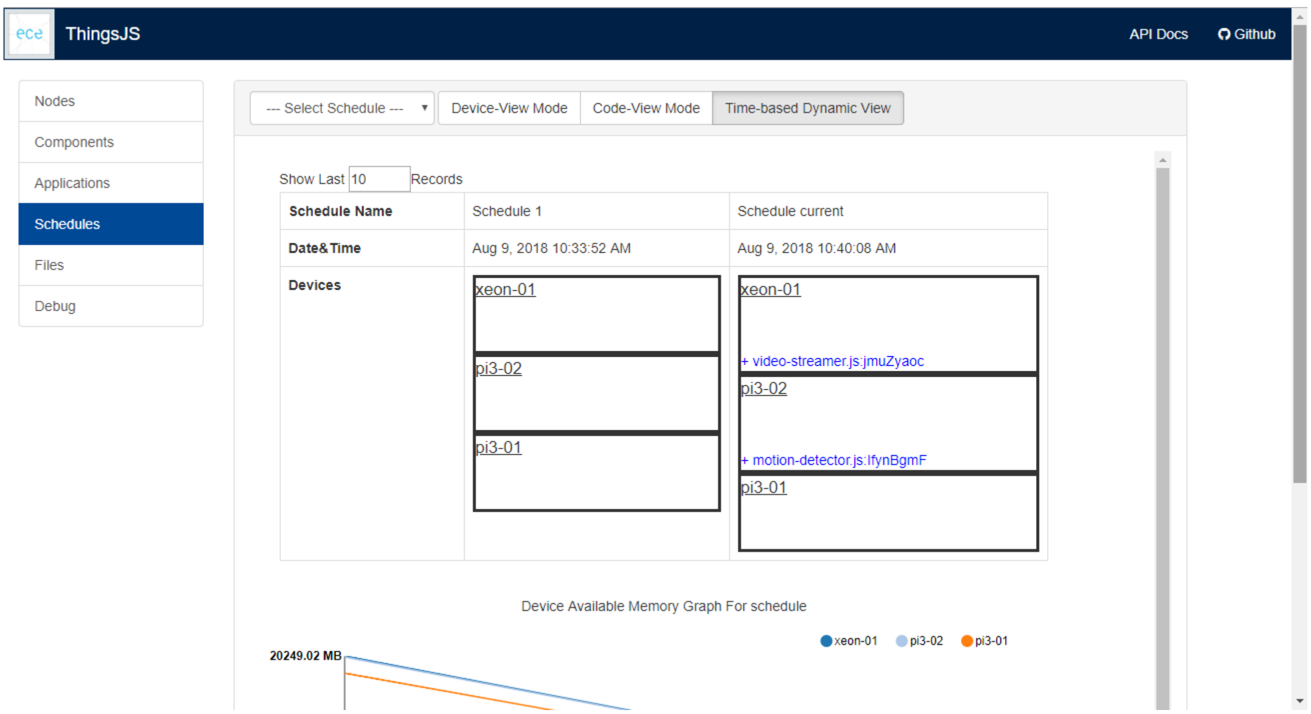
IoT components currently executing

Web Dashboard



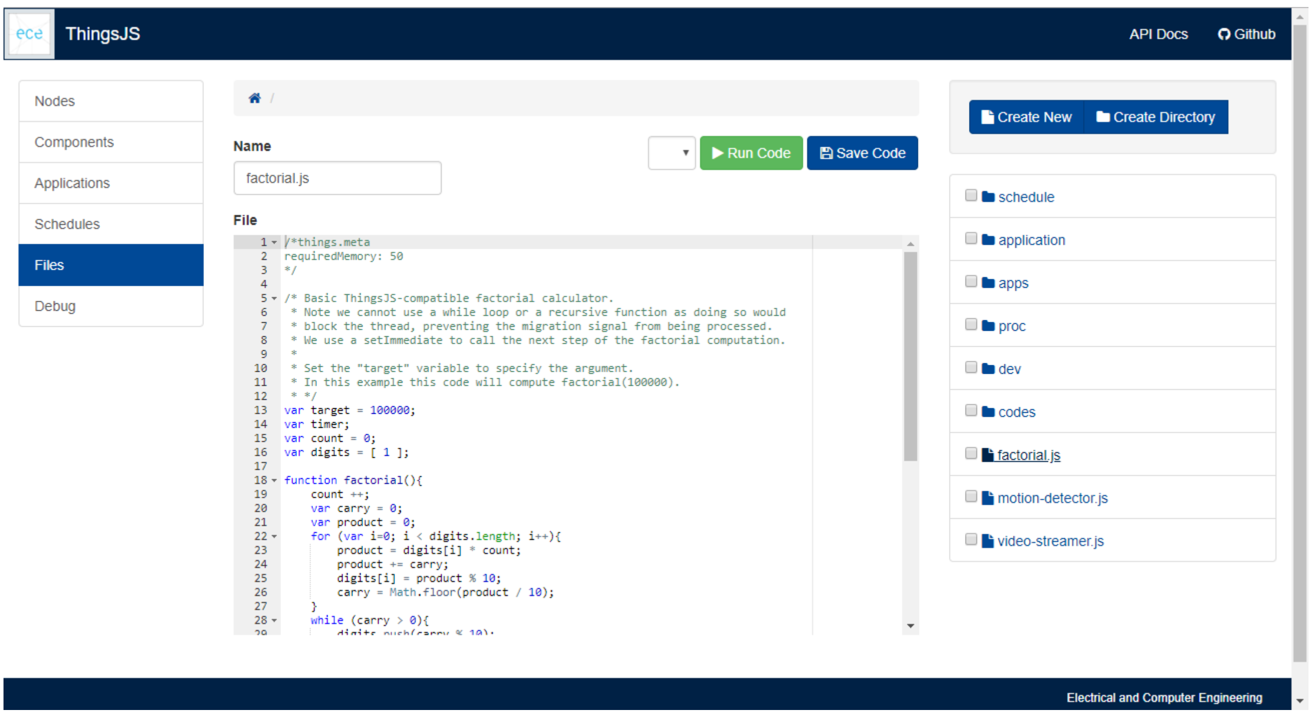
Applications: logical grouping of components

Web Dashboard



Scheduling of components to devices

Web Dashboard



File system and file/code editor

Table of Contents