Rusta: Elastic Processing and Storage at the Edge of the Cloud

Steffen Viken Valvåg,
Dag Johansen, Åge Kvalnes
University of Tromsø, Norway

HotTopiCS 2013
Prague, April 20, 2013

IAD INFORMATION ACCESS DISRUPTIONS

Background

- Cloud services are becoming ubiquitous
- Blurred distinction between traditional applications and services
  - Applications are expected to synchronize automatically across devices
- Proliferation of computing devices
  - Increased demand for mobility
  - Increased opportunity for offloading

Goals (I)

- Allow cloud services to flexibly integrate computing resources available on client devices and machines
  - **Processing**: Delegate work to freely available client machines rather than paying for processing time in the cloud
  - **Storage**: Since processing touches data, they are interlinked, and decentralized storage may be desirable
- Reduce operational costs, while preserving availability and fault tolerance

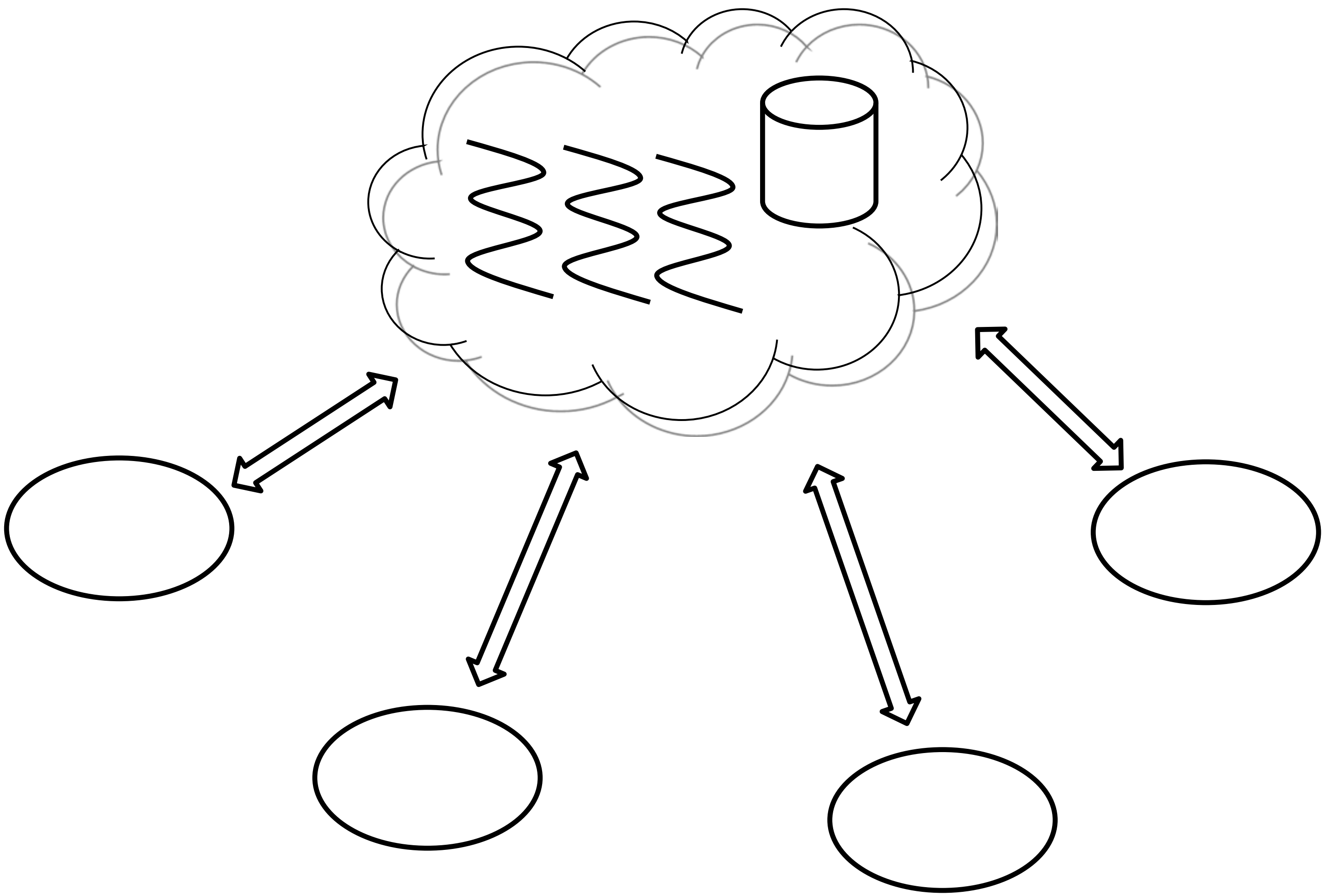
Goals (II)

- Allow traditional applications to seamlessly integrate computing resources available in the cloud
  - **Off-loading**: off-load work to the cloud to improve performance or preserve battery
  - **Synchronization**: Checkpoint and synchronize application state across multiple devices
- Decouple the mode of deployment from the application logic

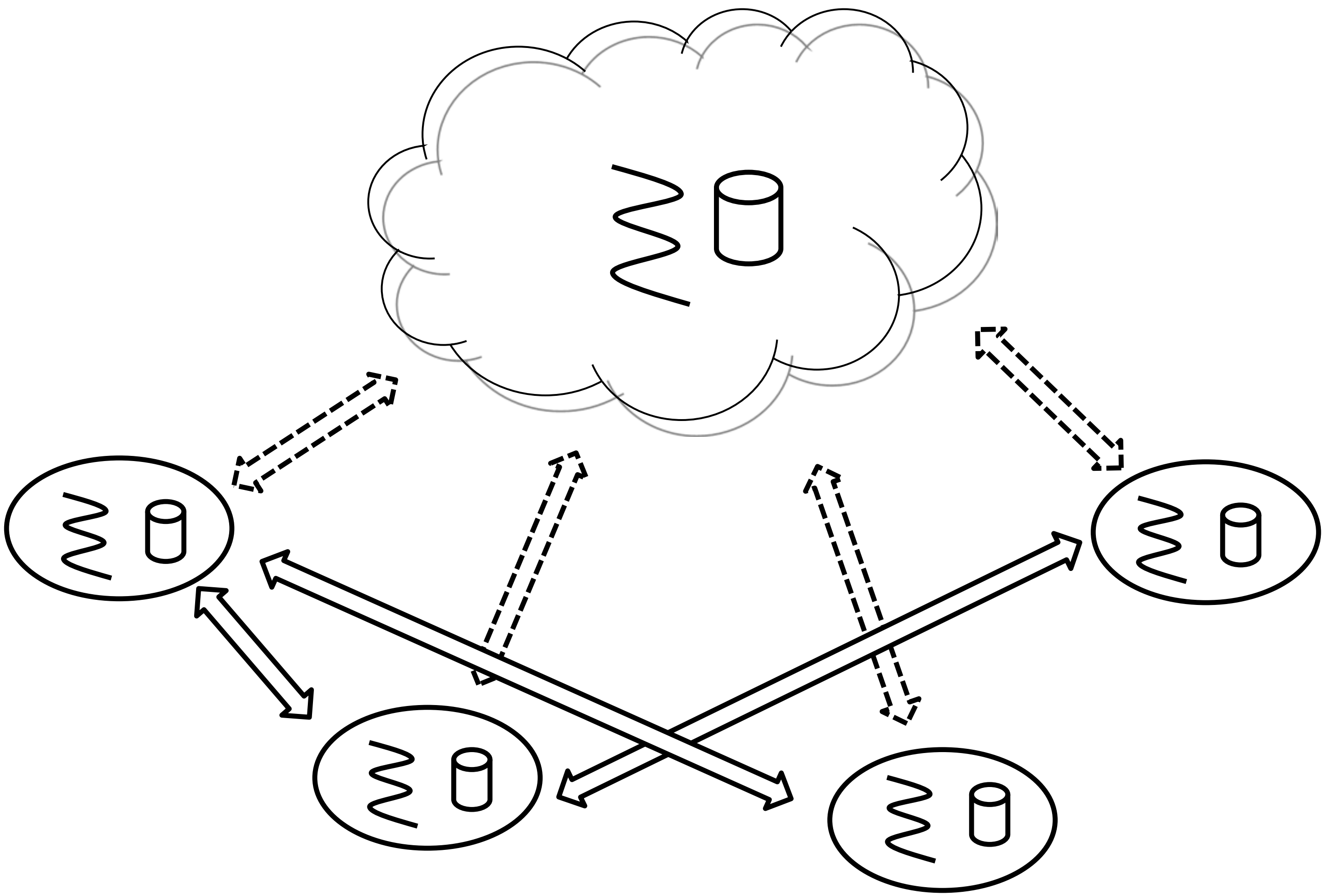
Rusta Architecture

- A centralized ***hub service*** is implemented as a conventional cloud service
  - Maintains critical system state
  - Common point of contact for all clients
  - Code repository (Java class files)
- Clients communicate sparingly with the hub
  - Bootstrapping new clients
  - Looking up other clients
  - Acquiring work to execute
  - To coordinate checkpoints

Traditional Service Architecture



Rusta Architecture



Implementation

- Hub service implemented on Google App Engine, in its Java servlet environment
  - Accessed via XMPP, i.e. using a chat client
  - “Command-line” admin interface via chat
- Client library implemented in Scala
  - Provides high-level programming abstractions for application/service developers
  - Uses Akka to drive an internal actor system
  - Communicates with the hub using GWT’s RPC
  - Polling using Google’s channel API

Scala

- High-level multi-paradigm language
  - Powerful type system
  - Flexible syntax
  - Structural pattern matching
  - **Closures**
- Runs on the Java virtual machine (JVM)
  - Integrates seamlessly with Java code
- Ideally suited for embedding domain-specific languages (DSLs)

Programming Interface

- Rusta clients are written in Scala
  - But can easily be glued to legacy Java code
- Functional continuation-based programming
  - But tailored to mimic an imperative style
- Location transparent
  - Data is accessed in a functional manner, so data locality can be arranged transparently
  - Move the computation to the data, or vice versa

Processes

- Rusta *processes* are light-weight execution units
  - Thread-like programming abstraction
  - Communicate through asynchronous message passing
  - Each process has a main message processing loop and may branch off into nested loops
- Process state is captured as a ***continuation closure***
  - Limited by heap space only; scales to hundreds of thousands of processes per machine
  - Easy to serialize and transfer process state

Hello World

- This process listens for a single system-generated *'start* message

```
object Example extends RustaApp {

  val example = new Deployment {
    group("mygroup") {
      process("myprocess") {
        case 'start => println("hello from " + me)
      }
    }
  }

  system.deploy(example)
}
```

Hello World

- This process listens for a single system-generated *'start* message

```
object Example extends RustaApp {

  val example = new Deployment {
    group("mygroup") {
      process("myprocess") {
        case 'start => println("hello from " + me)
      }
    }
  }

  system.deploy(example)
}
```

Closure that constitutes the main message handler

Hello World

- This process listens for a single system-generated *'start* message

```
object Example extends RustaApp {

  val example = new Deployment {
    group("mygroup") {
      process("myprocess") {
        case 'start => println("hello from " + me)
      }
    }
  }

  system.deploy(example)
}
```

Structural pattern matching

Hello World

- This process listens for a single system-generated *'start* message

```
object Example extends RustaApp {

  val example = new Deployment {
    group("mygroup") {
      process("myprocess") {
        case 'start => println("hello from " + me)
      }
    }
  }

  system.deploy(example)
}
```

Function of the Rusta API, taking curried arguments

Hello World

- This process listens for a single system-generated *'start* message

```
object Example extends RustaApp {

  val example = new Deployment {
    group("mygroup") {
      process("myprocess") {
        case 'start => println("hello from " + me)
      }
    }
  }

  system.deploy(example)
}
```

Process name and other optional arguments

Group Namespace

- Rusta *groups* define a hierarchical namespace for processes and data items
- Groups delineate sets of mutually trusted clients
  - Processes may migrate freely within their group
  - Data may be replicated freely within its group
- Allows fine-tuning trade-offs between privacy, availability, and elasticity
  - E.g., larger groups give more elasticity
- Potentially initialized from a social network

Message Passing

- Messages are arbitrary (immutable) objects
- Receivers use structural pattern matching to select (and parse) messages to process
- Senders may specify a reply handler
  - Specifies how to (eventually) process a reply
  - Messages are tagged with sequence numbers that are associated with pending reply handlers
  - Invoked asynchronously whenever a reply arrives
  - Syntactically tied to the sending code, improving legibility

Message Passing Example

```
group("images") {
  process("main") {
    case 'start => {
      send("thumbnails", ('get, "image.jpg")) {
        case null => println("No such image")
        case tn: Image => showImage(tn)
      }
    }
  }
}
```

Message Passing Example

```
group("images") {
  process("main") {
    case ’start => {
      send("thumbnails", (’get, "image.jpg")) {
        case null => println("No such image")
        case tn: Image => showImage(tn)
      }
    }
  }
}
```

Destination process

Message Passing Example

```
group("images") {
  process("main") {
    case 'start => {
      send("thumbnails", ('get, "image.jpg")) {
        case null => println("No such image")
        case tn: Image => showImage(tn)
      }
    }
  }
}
```

Message = arbitrary object
(a tuple in this case)

Message Passing Example

```
group("images") {
  process("main") {
    case ’start => {
      send("thumbnails", (’get, "image.jpg")) {
        case null => println("No such image")
        case tn: Image => showImage(tn)
      }
    }
  }
}
```

Reply handler

Data Access

- Conceptually similar to sending messages and handling replies
  - Lookups specify a data key to look up, and a closure to execute on the associated data value
- Just like the closures that capture process state, these closures can be serialized and transferred
  - Potentially sending the computation to the data
  - Maintains location transparency while encouraging data locality

```
group("images") {
  process("main") {
    case 'start => {
      send("thumbnails", ('get, "image.jpg")) {
        case null => println("No such image")
        case tn: Image => showImage(tn)
      }
    }
  }

  process("thumbnails") {
    case ('get, path: String) => {
      getData(path) {
        case (image: Image, thumbnail) => {
          reply(thumbnail)
        }
        case (image: Image, null) => {
          val tn = makeThumbnail(image)
          putData(path, (image, tn))
          reply(tn)
        }
        case null => reply(null)
      }
    }
  }
}
```

Look up a data item and process it asynchronously

```
group("images") {
  process("main") {
    case 'start => {
      send("thumbnails", ('get, "image.jpg")) {
        case null => println("No such image")
        case tn: Image => showImage(tn)
      }
    }
  }

  process("thumbnails") {
    case ('get, path: String) => {
      getData(path) {
        case (image: Image, thumbnail) => {
          reply(thumbnail)
        }
        case (image: Image, null) => {
          val tn = makeThumbnail(image)
          putData(path, (image, tn))
          reply(tn)
        }
        case null => reply(null)
      }
    }
  }
}
```

The context is preserved, so that the original message can still be replied to

Stateful Processes

- Close over variables in outer scopes
- The Scala compiler includes the variables in the closure's state
- Used for internal state, private to a process
  - E.g., for data aggregation

```
group("mygroup") {
  var x = 0

  process("myprocess") {
    case ‘getx => println("x = " + x); x += 1
  }
}
```

Ongoing Work

- Scheduling algorithms in the hub
  - Distribute processes among eligible clients
  - Process migration
- Data placement policies
  - Replication costs vs. availability
  - Cloud storage as a fallback to guarantee availability
- Checkpointing algorithms
  - Consistent cuts

Applications

- File sharing
  – Basic image sharing application
- Collaboration systems
- Multi-cloud services
- Lifelog image analysis
- Personal sensor data processing (soccer domain)
- Social networking
  – Analytics driven by client machines to empower users while preserving privacy

Summary

- Rusta allows flexible and decentralized deployment of cloud services
  - Alternatively: easy offloading from clients to the cloud, and to other clients
- Simple centralized architecture
  - But the central hub is not a bottleneck
- High-level programming interface in Scala
  - Light-weight and location-transparent processes
  - Easy to migrate (compared to e.g., thread migration)
- Process checkpointing and migration currently implemented
  - Basic image sharing application developed

Questions?