
Rustem Dautov
Iraklis Paraskakis
Dimitrios Kourtesis
Mike Stannett

20 April 2013
Motivation

- Flexibility of having cloud application platforms with built-in/third-party services comes with a cost
  - The complexity of service-based cloud environments is outgrowing our capacity to manage them in a manual manner
  - E.g., a failure of a notification service

- We need to automate the management process
  - Similar to the introduction of automatic branch exchanges in telephony in 1920s

- Self-management mechanisms have to be developed
  - A self-adaptation framework following the autonomic computing principle and based on IBM’s MAPE-K model
What do we need?

- Describe managed elements for self-reflection
- Encode critical condition patterns
- Homogenise representation of monitored data
- Assess situation and detect critical conditions
- Diagnose problems and reason about possible adaptation strategies
What are we monitoring?

• Data streams from multiple heterogeneous sources:
  • Deployed apps, platform components, 3rd-party services
An analogy

- From the IM perspective, cloud platforms are characterised by:
  - *Dynamism*: data is generated at an unpredictable rate
  - *Distributed nature*: data comes from logically and physically distributed sources
  - *Volume*: the amount of generated data is huge
  - *Heterogeneity*: data is heterogeneous in representation and/or semantics

- These characteristics are also shared by other problem domains
  - *Sensor Web*
Sensor Web

• **Sensor Network** – a computer accessible network of spatially distributed devices using sensors to monitor conditions at different locations, such as temperature, sound, pressure, etc.

• Sensor Web Enablement (SWE) project aims at developing a suite of specifications related to
  • Sensors
  • Sensor data models
  • Sensor Web services

that will be accessible and controllable via the Web.
A promising direction: Semantic Sensor Web (SSW)

- Addresses the challenges of SWE by utilising the Semantic Web technologies
- Enables situation awareness by providing enhanced meaning for sensor observations
  - E.g., RDF annotations to usual XML data
- A Sensor is anything that can calculate or estimate a data value:
  - An application component, an SQL query, a Web service, etc.
Semantic Web technologies

Querying over RDF streams: Continuous SPARQL Queries

Querying: SPARQL

Ontologies: OWL

Rules: RIF/SWRL

Taxonomies: RDFS

Data interchange: RDF

Syntax: XML

Identifiers: URI

Character set: UNICODE
Addressing the needs

- Describe managed elements for self-reflection
- Consume data from multiple heterogeneous sources
- Homogenise data representation
- Encode critical condition patterns
- Assess situation and detect critical conditions
- Diagnose problems and reason about possible adaptation strategies

- OWL ontologies
- RDF
- RDF Streams
- Continuous SPARQL Queries
- OWL
- SWRL
Conceptual architecture

![Conceptual Architecture Diagram]

- **Managed elements**
  - Apps/platform components
  - Raw data
  - Triplification engine
- **Execution**
  - Adaptation actions
  - Knowledge
    - OWL ontologies/SWRL rules
- **Analysis/Planning**
  - Continuous SPARQL query engine
  - Query results
  - OWL/SWRL reasoning engine
- **Monitoring**
  - RDF triples
- **Autonomic manager**

HotTopiCS 2013, Prague, Czech Republic, 20-21 April 2013
Simple use case

• A number of applications are deployed on a cloud platform and rely on a built-in notification service.
• The notification service gets overloaded and cannot process all incoming requests.
• We need to detect such situations and switch over some of the dependent applications to an external substitute.
  • Response time threshold = 5 sec
• The prototype autonomic manager was deployed on CloudFoundry and developed in Java Spring using:
  • OWL API library
  • C-SPARQL library
Prototype implementation

1. The client sends a request to the Web service
2. The Web service sends a reply to the client
3. The client calculates the response time and sends an RDF triple to the RabbitMQ queue
4. The AM receives RDF triples from the stream and performs querying by means of the C-SPARQL engine
Sample OWL ontology

```owl
@prefix : <http://www.seerc.org/ontology.owl#> .


:Service rdf:type owl:Class .

:Time rdf:type owl:Class .

:hasResponseTime rdf:type owl:ObjectProperty ,
    rdfs:domain :Time .

:isEquivalent rdf:type owl:ObjectProperty ,
    owl:SymmetricProperty ;
    rdfs:range :Service ;
    rdfs:domain :Service .

:hasHighResponseTime rdf:type owl:DatatypeProperty ,
    rdfs:range xsd:Boolean .

:hasValue rdf:type owl:DatatypeProperty ,
    rdfs:range xsd:int .

:needsSubstitution rdf:type owl:DatatypeProperty ,
    rdfs:range xsd:Boolean .
```
Sample RDF stream

- Each RDF triple represents a change in response time from a service and annotated with a timestamp.
- The sample stream represents a sudden increase in a service’s response time.

```
@prefix ex: <http://www.seerc.org/ontology/>

ex:#Service1 ex:hasResponseTime; 1000. [2012-09-18 13:24:52]
ex:#Service1 ex:hasResponseTime; 890.  [2012-09-18 13:24:54]
ex:#Service1 ex:hasResponseTime; 1110. [2012-09-18 13:24:56]
ex:#Service1 ex:hasResponseTime; 1300. [2012-09-18 13:24:58]
ex:#Service1 ex:hasResponseTime; 5450. [2012-09-18 13:25:13]
ex:#Service1 ex:hasResponseTime; 6000. [2012-09-18 13:25:20]
ex:#Service1 ex:hasResponseTime; 6700. [2012-09-18 13:26:15]
```
Sample C-SPARQL query

- The sample C-SPARQL query is registered against a data stream and triggers whenever response time from a service exceeds 5000 ms.

```sparql
PREFIX ex:<http://www.seerc.org/ontology/>
SELECT DISTINCT ?service
FROM STREAM http://www.seerc.org/stream
    [RANGE 60s STEP 1s]
WHERE { ?service ex:hasResponseTime ?time .
    FILTER (?time > 5000) }
```
Sample SWRL rules

Rule 1: Has high response time

Service(?s1) ^ Time(?t) ^
hasResponseTime(?s1, ?t) ^
greaterThan(?t, 5000) ->
hasHighResponseTime(?s1, true)

Rule 2: Needs substitution

hasHighResponseTime(?s1, true) ^
Service(?s2) ^ isEquivalentTo(?s1, ?s2)
-> needsSubstitution(?s1, ?s2)
Results

• We can:
  • monitor response time from services;
  • detect whether response time from a service is exceeding its threshold
  • generate a diagnosis and suggest an adaptation strategy

• Further experiments:
  • Scalability
  • Portability across several application platforms
  • Accuracy
# Initial experiments

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Request frequency</th>
<th>Number of queries</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>1 request/sec</td>
<td>1</td>
<td>Critical conditions detected within 1 second</td>
</tr>
<tr>
<td>100 threads</td>
<td>1 request/sec</td>
<td>1</td>
<td>Critical conditions detected within 1 second</td>
</tr>
<tr>
<td>500 threads</td>
<td>1 request/sec</td>
<td>1</td>
<td>The client application crashes due to the limitation of 512 MB RAM</td>
</tr>
<tr>
<td>1 thread</td>
<td>1 request/sec</td>
<td>1000</td>
<td>Critical conditions detected within 1 second</td>
</tr>
<tr>
<td>1 thread</td>
<td>1 request/sec</td>
<td>5000</td>
<td>The monitor crashes due to the limitation of 512 MB RAM</td>
</tr>
<tr>
<td>400 threads</td>
<td>1 request/sec</td>
<td>4000</td>
<td>Critical conditions detected within 1 second</td>
</tr>
</tbody>
</table>
Further work

• Defining evaluation and testing strategies
  • Important aspects: scalability, flexibility, analysis support, performance

• Further developing and experimenting
  • Demonstrating the “reasoning power” of the approach
  • Porting the framework to OpenShift and AppScale
  • Extending the monitoring scope to several parameters
Thank you!

Questions?