Challenges in Data Stream Processing

Corso di Sistemi e Architetture per Big Data
A.A. 2017/18

Valeria Cardellini

Challenge 1: Optimize the DSP application

- Apply some transformation to streaming graph
  - At design time or run-time
- Operator reordering
  - To avoid unnecessary data transfers

• Redundancy elimination
Challenge 1: Optimize the DSP application

- Operator separation

\[ A \rightarrow A_1 \rightarrow A_2 \]

- Operator fusion

\[ A \rightarrow B \rightarrow AB \]

- Operator fission (i.e., data parallelism)

\[ A \rightarrow Split \rightarrow A \rightarrow Merge \]
At the streaming system layer

- The previous challenge is addressed at the DSP application layer
- What about the streaming system layer?
- Two main classes of solutions to improve performance (e.g., to control application latency) at the streaming system layer
  1. Place the operators
  2. Manage load variations

Challenge 2: Place the operators

- Determine, within a set of available distributed computing nodes, those nodes that should host and execute each operator instance of a DSP application
Challenge 2: Place the operators

• Operator placement decision: a complex problem
  – Trade communication cost against resource utilization

• When
  – Initial (static) operator placement
    • Can be more expensive and comprehensive
  – Can also be at run-time
    • Place again all the operators or only a subset
    • Require self-adaptation

• We will focus on this issue later

Challenge 3: Manage load variations

• Typical stream processing workloads are:
  – with high volume and high rates
  – bursty and with workload spikes not known in advance
    • Twitter in 2013: rate of tweets per second = 5700
    • … but significant peak of 144,000 tweets per second
Challenge 3: Manage load variations

- Some solutions:
  - Admission control
  - Static reservation
  - Reserve specific resources in advance
  - Cons: over-provisioning and cost increase
  - Apply dynamic techniques such as load shedding
  - Selectively drop tuples at strategic points (e.g., when CPU usage exceeds a specific limit)
  - Cons: sacrifice accuracy and completeness

A Shedder A

- Some solutions (continued):
  - Use adaptive rate allocation
    - E.g., backpressure: the upstream operator that precedes the bottleneck stores data in an internal buffer to reduce the pressure; backpressure recursively propagates up to the source operators
  - Redistribute load, e.g., determine new operator placement and relocate operators on computing nodes
    - Cons: available resources could be insufficient

- What else?
Exploit elasticity

• Another solution:
  – Detect bottleneck and solve it by exploiting **elasticity**: acquire and release resources when needed

  ![Chart showing resources vs time]

  – *How?*
    • By hand: possible, but cumbersome
    • So what? MAPE!

Elastic data stream processing

• **Where?**
  – At **application layer** (i.e., data parallelism)
    • i.e., apply **SPMD** paradigm: concurrent execution of multiple replicas of the same operator on different data portions
    • Scale out (in) operators by adding (removing) operator replicas

  ![Diagram of SPMD paradigm]

  – At **infrastructure layer**
    • Scale horizontally (scale-out/in) computing resources (containers, virtual machines, physical machines)
    • Also vertical scaling (containers, virtual machines)
Elastic stream processing

• *When* and *how* to scale?
  – Open issues
  – Some simple example:
    • When: threshold-based (like AWS Auto Scaling)
    • How: add/remove one operator replica at time
    • Where: determine randomly the location of the new replica

• Be careful: elasticity overhead is not zero!
  – In most streaming systems: run a new placement decision to take the new replicas into account
  – Dynamic scaling impacts stateful operators

Challenge 4: Self-adapt at run-time

• Many factors may change at runtime, e.g.,
  – Load variations, QoS of computing resources, cost of computing resources (e.g., due to dynamic pricing schemes), network characteristics, node mobility, …

• How to adapt the DSP application when changes occur?
  – Enrich DSP systems with run-time adaptation capabilities

• Which *adaptation actions*?
  – Relocate the operators through migration
  – Scale-out/in the number of operator instances
Self-adaptive deployment

• **MAPE** *(Monitor, Analyze, Plan and Execute)*

• **Plan** phase: how to **reconfigure** the DSP application deployment

---

**Distributed Storm**

• We developed an extension of Storm
• **Goals:** to provide
  – distributed monitoring
  – distributed placement
  – and adaptation capabilities
• **Where:** large-scale environment
• **Code available on GitHub**
  [matnar.github.io/uniroma2-storm/](http://matnar.github.io/uniroma2-storm/)

Distributed Storm architecture

Distributed Storm: monitoring

- **QoSMonitor** (for each worker node)
  - Estimate network latencies
    - Use a network coordinate system
    - Vivaldi’s algorithm: decentralized and gossip-based
  - Monitor QoS attributes
    - Node utilization and availability

- **Worker Monitor** (for each worker process)
  - Monitor exchanged data rate among the operators
Distributed Storm: performance

But distributed placement suffers from lack of coordination

- We compared fully distributed placement heuristic implemented in Distributed Storm (Pietzuch et al.) with our optimal placement policy (ODP)
Reconfiguration challenges

- Reconfiguring the deployment has a non negligible cost!
- Can affect negatively application performance in the short term
  - Application freezing times caused by operator migration and scaling, especially for stateful operators

Perform reconfiguration only when needed

Take into account the overhead for migrating and scaling the operators

Challenge 5: stateful operators

- State complicates things…
  1. Dynamic scaling
  2. Operator re-placement
  3. Recovery from failure

Impact state

Loss of state!
Approaches for stateful migration

• Most streaming systems do not support stateful processing and migration (e.g., Storm)
  – Developers manage state
  – Typically combined with external system to store state
  – Increased design complexity

• Recent interest in research prototypes and streaming systems (e.g., Heron, Spark Streaming)

• Requirements for stateful operator migration
  – Safety (i.e., to preserve the consistency of the operations)
  – Application transparency
  – Minimal footprint

Issues with stateful operators

• Require mechanisms to:
  – Migrate stateful operators
    • Pause-and-resume approach
    • Parallel track approach
  – Partition streams and load balance among replicas
Stateful operator migration

- Pause-and-resume approach
  - Application latency peak during migration

  1. Stop migrating task
  2. Save state
  3. Terminate migrating task and start it on new node
  4. Restore state
  5. Resume stream processing

- Parallel track approach
  - Old and new operator instances run concurrently until their state is synchronized
    - No latency peak
    - Enhanced mechanisms for synchronization
Issues for stateful migration

• How to identify the portion of state to migrate?
  – Expose an API to let the user manually manage the state
  – Support only partitioned stateful operators
    • Partitioned stateful operators store independent state for each sub-stream identified by a partitioning key
    • Automatically determine, on the basis of a partitioning key, the optimal number of state partitions to be used and migrate

• How to balance the load among multiple stateful replicas?
  • Can use consistent hashing
  • Can use partial key grouping
    – Uses two hash functions where a key can be sent to two different replicas instead of one
  • Only available in research prototypes
Elastic stateful migration in Storm

- We developed mechanisms for elasticity and stateful migration in Storm

- Migration protocol based on pause-and-resume approach to relocate the operator internal state on a different node

- Elasticity policy at the application level
  - Simple threshold-based policy


Valeria Cardellini - SABD 2017/18
Elastic distributed DSP in Storm

- We augmented Distributed Storm with MAPE capabilities and optimal centralized placement and reconfiguration policy that keeps into account reconfiguration costs.

Elastic distributed DSP in Storm

- Query 1 of DEBS’15 Grand Challenge

- We proposed EDRP, a flexible approach to model the reconfiguration cost.


Valeria Cardellini - SABD 2017/18
Still some limitations

- Centralized optimization algorithms do not scale for large problem instances
- Centralized MAPE architecture does not scale in a geo-distributed environment
  - Distributed components but logic still centralized
  - Fully distributed solutions have limitations
- Which solution?
  **Decentralize MAPE**

Easier said than done!

- Many patterns for decentralized control
  - Each one with pros and cons

Our choice
Hierarchical MAPEs in Storm

- Augmented Distributed Storm with MAPE capabilities and elasticity control

Challenge 6: guarantee fault tolerance

- DSP applications run for long time
  failures are unavoidable
- Possible solutions:
  - Active replication
  - Check-pointing
  - Replay logs
  - Hybrid solutions
- Having different trade-offs between runtime cost in absence of failures and recovery cost
- Large-scale complicates things…
  - Network partitions and CAP theorem
References


• M. de Assuncao, A. da Silva Veith, R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions", *J. of Network and Computer Applications*, 2018. [https://hal.inria.fr/hal-01653842/document](https://hal.inria.fr/hal-01653842/document)
