Toward large-scale distributed stream processing: models, systems and challenges

Valeria Cardellini and Francesco Lo Presti
University of Rome Tor Vergata, Italy

2nd Int’l Summer School on Autonomous Control for Reliable Future Networks and Services, 30 May 2016, Opatija, Croatia
Who are we?

Valeria Cardellini  
Associate professor  
@ Univ. of Rome Tor Vergata

Francesco Lo Presti  
Associate professor  
@ Univ. of Rome Tor Vergata

• Joint research work with Vincenzo Grassi and Matteo Nardelli
The data deluge

• Some well-known numbers related to Big Data:
  – Every day in 2014 we created 2.5 Exabytes
  – 40 Zettabytes of data will be created by 2020

• Proliferation of new sources of data
  – Sensors, mobile devices, cameras
  – Social networks
  – Scientific instruments
  – Vehicles

• How can we make sense of all these data?
  – Process data to extract valuable insights
Why data stream processing?

• Applications such as:
  – Sentiment analysis on multiple tweet streams @Twitter
  – User profiling @Yahoo!
  – Tracking of query trend evolution @Google
  – Fraud detection
  – Bus routing management @city of Dublin [Art14]

• Require:
  – Continuous processing of unbounded data streams
    generated by multiple, distributed sources
  – In (near) real-time fashion
Why data stream processing?

• In the past years data stream processing (DSP) was considered a solution for very specific problems (e.g., financial tickers)

• But now we have (and will have) more general settings
  – E.g., Internet of Things
Why data stream processing?

• Decrease the overall latency to obtain results
  – No data persistence on stable storage
  
  See “Latency numbers every programmer should know”

<table>
<thead>
<tr>
<th></th>
<th>Main memory reference: 100ns</th>
<th>Send 2,000 bytes over commodity network: 177ns</th>
<th>Read 1,000,000 bytes sequentially from SSD: 123,000ns ≈ 123μs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 cache reference:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch mispredict:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 cache reference:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mutex lock/unlock:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17ns</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000ns ≈ 1μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000ns ≈ 10μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100,000ns = 1ms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V. Cardellini - ACROSS 2nd Summer School
Why data stream processing?

• Decrease the overall latency to obtain results
  – No data persistence on stable storage
    See “Latency numbers every programmer should know”
  – No periodic batch analysis

• Simplify the data infrastructure

• Make time dimension of data explicit
Traditional DSP challenges

• Stream data rates can be high and data arrive in large volumes
  – High resource requirements for processing (clusters, data centers, distributed Clouds)

• Processing stream data has real-time aspects
  – Stream processing applications have QoS requirements, e.g., end-to-end latency
  – Must be able to react to events as they occur
Why large-scale stream processing?

• Goals: increase scalability and reduce latency
• How? Rely on distributed and near-edge computation
Goals of the lectures

• Give a flavor of large-scale distributed stream processing and related research challenges

• *Part I* (V. Cardellini)
  – Focus on *system issues*
  – These slides

• *Part II* (F. Lo Presti)
  – Focus on *models and algorithms*

• *Request*
  – If you get either bored or lost, ask questions...
  – If you like to ask questions, ask questions...
Goals of the lectures

• Give a flavor of large-scale distributed stream processing and related research challenges

• *Part I* (V. Cardellini)
  – Focus on *system issues*

• *Part II* (F. Lo Presti)
  – Focus on *models and algorithms*

• *Request*
  – If you get either bored or lost, ask questions...
  – If you like to ask questions, ask questions...
Data stream definitions
Data stream

• "A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety.

Queries over streams run continuously over a period of time and incrementally return new results as new data arrive." [Gol03]
Sliding windows

- How many data items should we process each time?
  - Process items in window-sized batches
    - *Count-based* window, e.g., last $n$ items
    - *Time-based* window, e.g. from $[t-T]$ to $[t]$
Sliding windows

• How often should we evaluate the window?
  – **Eager approach**: output new result items as soon as available (but can be difficult to implement efficiently)
  – **Lazy approach**: slide window by $s$ seconds (or $m$ items)
DSP application model

• A DSP application is made of a network of operators (processing elements) connected by streams, at least one data source and at least one data sink

• Represented by a directed graph
  – Graph vertices: operators
  – Graph edges: streams

• Graph can be cyclic
  – Some systems only support directed acyclic graph (DAG)

• Graph topology rarely changes
DSP operator

- A self-contained processing element that:
  - transforms one or more input streams into another stream
  - can execute a generic user-defined code
    - Algebraic operation (filter, aggregate, join, ..)
    - User-defined (more complex) operation (POS-tagging, ..)
  - can execute in parallel with other operators

- Can be stateless or stateful
  - **Stateless**: know nothing about the state (e.g., filter, map)
  - **Stateful**: keep some sort of state
    - E.g., some aggregation or summary of processed elements, or state-machine for detecting patterns for fraudulent financial transaction
    - State might be shared between operators
“Hello World”: WordCount
Some DSP application: DEBS’14 GC

• Real-time analytics over high volume sensor data: analysis of energy consumption measurements [DEBS14GC]
  – Smart plugs deployed in households and equipped with sensors that measure values related to power consumption
    • Input data stream:
      2967740693, 1379879533, 82.042, 0, 1, 0, 12

• Query 1: make load forecasts based on current load measurements and historical data
  – Output data stream:
    ts, house_id, predicted_load

• Query 2: find the outliers concerning energy consumption
  – Output data stream:
    ts_start, ts_stop, household_id, percentage
Some DSP application: DEBS’15 GC

- Real-time analytics over high volume spatio-temporal data streams: analysis of taxi trips based on data streams originating from New York City taxis [DEBS15GC]

  - Query 1: identify recent frequent routes
  - Query 2: identify regions with the highest profit
  - Both queries rely on a sliding window operator
    - Continuously evaluate the query results

- Use geo-spatial grids to define the events of interest
Some DSP application: DEBS’16 GC

- Real-time analytics for a dynamic (evolving) social-network graph [DEBS16GC]
- *Query 1*: identify the posts that currently trigger the most activity in the social network
- *Query 2*: identify large communities that are currently involved in a topic
- Require continuous analysis of dynamic graph considering multiple streams that reflect graph updates
Data stream systems
Streaming system

• Distributed system that executes stream graphs
  – continuously calculates results for long-standing queries
  – over potentially infinite data streams
  – using operators
    • that can be stateless or stateful

• System nodes may be heterogeneous

• Must be highly optimized and with minimal overhead so to deliver real-time response for high-volume DSP applications
Operator placement

• Determine, within a set of available distributed computing nodes, the nodes that should host and execute each operator of a DSP application
Big data centers

• Which frameworks for data stream processing?
• Usually run in locally distributed clusters within large data centers

• Assumptions:
  – Scale out and not scale up
    • Commodity servers
    • Data-parallelism is king
  – Software designed for failure
    • See [Dea09]

Source: Google
Apache Storm

- Apache Storm
  - Open-source, real-time, scalable streaming system
  - Provides an abstraction layer to execute DSP applications

- **Topology** (streaming graph)
  - Spouts (data sources) and bolts (operators and data sinks)
Storm entities

- **Task**: operator instance
- **Executor**: smallest schedulable entity
  - Execute one or more tasks related to same operator
- **Worker process**: Java process running a subset of executors
- **Worker node**: computing resource, a container for worker processes
Storm architecture
Other frameworks *(partial list)*

• Cloud-based frameworks
  – Amazon Kinesis
  – Google Cloud Dataflow
  – Microsoft

• Apache Spark
  – Improve MapReduce *(batch processing)*
  – Spark Streaming: reduce the size of each stream and process streams of data *(micro-batch processing)*
Other frameworks (*partial list*)

• Cloud-based frameworks
  – Amazon Kinesis
  – Google Cloud Dataflow
  – Microsoft

• Apache Spark
  – Improve MapReduce (*batch processing*)
  – Spark Streaming: reduce the size of each stream and process streams of data (*micro-batch processing*)
Other frameworks *partial list*

- **Cloud-based frameworks**
  - Amazon Kinesis
  - Google Cloud Dataflow
  - Microsoft

- **Apache frameworks**
  - Im
  - Spark

<table>
<thead>
<tr>
<th></th>
<th>One-at-a-time (e.g., Apache Storm)</th>
<th>Micro-batched (e.g., Apache Spark)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lower latency</td>
<td>![Green check]</td>
<td>![Green check]</td>
</tr>
<tr>
<td>Higher throughput</td>
<td>![Green check]</td>
<td>![Green check]</td>
</tr>
<tr>
<td>At-least-once semantics</td>
<td>![Green check]</td>
<td>![Green check]</td>
</tr>
<tr>
<td>Exactly-once semantics</td>
<td>![Green check]</td>
<td>![Green check]</td>
</tr>
<tr>
<td>Simpler programming model</td>
<td>![Green check]</td>
<td>![Green check]</td>
</tr>
</tbody>
</table>

V. Cardellini - ACROSS 2nd Summer School
A new breadth of frameworks

- Lambda architecture
  - Data-processing design pattern to handle massive quantities of data and integrate batch and real-time processing within a single framework

Source: https://voltdb.com/products/alternatives/lambda-architecture

V. Cardellini - ACROSS 2nd Summer School
Challenges in data stream processing
Challenge 1: Optimize the DSP application

- Apply some transformation to streaming graph
  - At design time or run-time
- Operator reordering [Hir14]
  - To avoid unnecessary data transfers
- Redundancy elimination [Hir14]
Challenge 1: Optimize the DSP application

- **Operator separation** [Hir14]

  ![Operator separation diagram]

- **Fusion** [Hir14]

  ![Fusion diagram]
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
    • Move only relocatable operators
    • Require operator migration

• See Part II
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

• Possible approaches:
  – 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
Challenge 3: Manage load variations

• Possible approaches (continued):
  – Use adaptive rate allocation [Bou12]
  – Redistribute load, e.g., determine new operator placement and relocate operators on computing nodes
    • Cons: available resources could be insufficient
Exploit data parallelism

• Alternative solution:
  – Detect bottleneck
  – Use **data-parallelism** (aka *operator fission* [Hir14])
    • Apply **SIMD** paradigm: concurrent execution of multiple replicas of the same operator on different data portions
    • By hand: possible, but cumbersome
Elastic stream processing

• Exploit **elasticity**: acquire and release resources when needed

  – At **application layer** (i.e. data parallelism)
    • Scale out (or scale in) operators
    • Activate (or deactivate) replicated operators [Bel14]

  – At **infrastructure layer**
    • Scale out (or scale in) computing nodes
Elastic stream processing

• *When* and *how* to scale?
  – See *Part II*

• But elasticity overhead is not zero!
  – In most streaming systems: run a new placement decision to take the new resources into account
  – Dynamic scaling impacts stateful operators
Challenge 4: Self-adapt at run-time

• To cope with highly dynamic operative environment
  – Unpredictable workload
  – Computational characteristics of operators not known a-priori
  – Need to sustained load for long provisioning times
  – Node availability, network congestion, ...
• Exploit run-time adaptation capabilities of streaming systems
• What adaption actions?
  – Scale the number of operator instances, relocate the operators, ...
Self-adaptation framework

• **MAPE**: Monitor, Analyze, Plan and Execute
• Software reference framework for self-adaptation
Distributed Storm

• We developed an extension of Storm [Car15]
• Goals: to provide
  – distributed monitoring
  – distributed placement (see Part II)
  – and adaptation capabilities
• Where: large-scale environment
• Code available on GitHub
  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 [ref]: 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

Load spike on a subset of nodes

~50%
Self-adaptation challenges

• Adaptation has a non negligible cost!
  – Run-time reconfigurations can increase latency and reduce application availability
    • Perform adaptation only when needed
  – Costs of operator migrations cannot be neglected
    • Freeze times caused by operator migration
    • How to migrate stateful operators?
Challenge 5: stateful operators

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

Loss of state!
Approaches for stateful migration

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

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

- Parallel track approach [Hei14]
- Pause-and-resume approach

![Diagram showing stateful operator migration process]

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

• How to identify the portion of state to migrate?
  – Expose an API to let the user manually manage the state [Fer13]
  – Support only partitioned stateful operators [Ged14]
    • 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
Elastic stateful migration in Storm

• We developed mechanisms for elastic stateful migration in Storm [Car16]

• Code on GitHub matnar.github.io/elastic-storm/

V. Cardellini - ACROSS 2nd Summer School
Elastic stateful migration in Storm

• Scaling decisions at the framework level
  – Adapt the number of parallel instances for each application operator
  – Simple threshold-based scaling policy (see *Part II*)

• Relocate the operator internal state on a different node and enable Storm to change the application deployment at run-time
Performance results

- DSP application: frequent pattern detection

  - Elastic scaling and stateful migration improve the application latency
Challenge 6: guarantee fault tolerance

- DSP applications run for long time intervals, failures are unavoidable
- Possible solutions:
  - Active replication [Bri09]
  - Check-pointing [Seb11]
  - Replay logs [Bal08]
  - Hybrid solutions [Zha10]
- Having different trade-offs between runtime cost in absence of failures and recovery cost
- Large-scale complicates things...
  - Network partitions and CAP theorem
Challenge 7: Manage multiple concurrent DSP applications

• Consider multiple competing DSP applications
• How should the streaming system allocate resources?
  – Fairness
  – Resource utilization
  – Profitability, ...
Apache Mesos

• Run concurrent frameworks on the same cluster and dynamically share the cluster resources

• Mesos: a cluster “operating system” [Hin11]
  – Efficient resource isolation and sharing across distributed frameworks
Apache Mesos

- Two-level scheduling based on Dominant Resource Fairness (DRF) algorithm
GMesos: distributed Mesos

- We are currently developing GMesos for large-scale environment... stay tuned!
Some new challenges and research opportunities

• Integrate data stream processing with SDN
  – With SDN, network into the control loop

• Study cross-layer optimization

• Address security and privacy issues in data stream processing
References


References


References


Thank you! Any questions?

Valeria Cardellini
cardellini@ing.uniroma2.it
www.ce.uniroma2.it/~valeria