Adaptive Rate Allocation in Distributed Stream Processing Systems under Bursty Workloads

Giannis Boutsis and Vana Kalogeraki
Department of Informatics
Athens University of Economics and Business
{mpoutsis,vana}@aueb.gr
On-Line Data Processing Systems

Real-time sensor data analysis system

Environmental monitoring applications

Customization of multimedia or news feeds

Network traffic monitoring

mpoutsis@aueb.gr
Distributed Stream Processing Systems

- In DSPS’s, data produced by large numbers of globally-distributed data sources is composed dynamically and processed to generate results of interest.

High volume data streams (sensor data, financial data, media data)

Real-time online processing functions/Continuous query operators

Extracted result streams

mpoutsis@aueb.gr
Streams are processed online by components distributed across hosts.

Data arrive in large volumes and high rates, while workload spikes are not known in advance.

Stream processing applications have QoS requirements, e.g., e2e delay.

Vulnerable to great variations in processing and communication delays.
Synergy

Distributed Stream Processing Application

Middleware

Physical Network

mpoutsis@aueb.gr
Synergy modules

Main modules:

- **Discovery module** for identifying application components and data streams
- **Routing module** for routing data streams and protocol messages between nodes
- **Monitoring module** for building resource utilization profiles
- **Composition module** that implements a fully distributed composition protocol that selects and instantiates application components so that QoS requirements are met.

mpoutsis@aueb.gr
Synergy modules (cont)

- **Application module** that implements the application logic

- **Replica placement module** that determines the replication and placement of component replicas on the Synergy nodes to maximize application availability

- **Scheduling module** that implements various scheduling algorithms on the Synergy nodes

- We extend Synergy middleware with burst management components that attempt to dynamically determine the rate allocation in order to react to the bursts
Our problem

• Data arrives in bursts
• Applications have real-time QoS demands (e2e time, rates etc)
• Workload not known in advance
  – High workload on a node affects all of its applications

• Our goal:
  – Accommodate bursts by adaptive rate allocation techniques
  – Meet QoS demands
Current Techniques

- **Static Reservation**
  (reserves specific resources in advance)

- **Load Shedding**
  (drops packets when CPU usage exceeds a specific limit)
System Model

- Applications: $A_i$
- Components: $C_i$
- Data units

User submits a request for:
- a set of applications $\{appq\}, 1 \leq q \leq Q$
- initial rate requirements $r_q$

- Application $A_i$ is characterized by its deadline: $Deadline_q$
- Component $C_i$ is characterized by its resource requirements $CPU_{ci}(j)$ for each resource $j$ and its selectivity $Sel_{Ci}$
General Optimization Problem

• Our main objective is to maximize the rates in the entire system with respect to the time and resource constraints.

\[
\text{maximize } \sum_{c_i \in C} W_{c_i} \ast R_{c_i}
\]

where: \( W_{c_i} = 10^{Priority_{c_i}} \)

priority can be set as 1, 2 or 3 for low, medium and high
Execution Time

\[
ExecTime_q = \max_{path} \left( \sum_{e_i \in p} Comp_{c_i}(R_q) \cdot CPU_{c_i} + \sum_{e_i \in p} Comm_{e_i \rightarrow e_{i+1}}(R_q) \right)
\]

\[
Comp_i(R_q) = C_i \cdot R_{c_i}
\]
Constraints

• Target Time Constraint: \( \text{ExecTime}_q \leq \text{Deadline}_q \)

• Resource Constraint: \( \sum_{c_i \in n} R_{c_i} \times (\text{CPU}_{c_i} \times \frac{C_i}{\text{Deadline}_q}) \leq 1 \quad \forall n \in \text{Nodes}_q \)

• Rates Constraint: \( \text{MinRate}_{c_i} \leq R_{c_i} \leq \text{MaxRate}_{c_i} \quad \forall c_i \in \text{Components} \)

• Flow conservation Constraint: \( R_{c_{i+1}} = \text{sel}_{c_i} \times R_{c_i} \quad \forall c_i \in \text{Components} \)
Linear Programming (LP) Formulation

• Simplex algorithm is a very popular method for linear programming problems
• Two main disadvantages:
  – Not scalable
  – Needs to be computed centralized

• Our implementation is based on the work by Van Slyke where a scalable and distributed algorithm for Simplex was presented
• The distributed algorithm
  – Seems to be faster than original Simplex that runs centralized
  – Allows every component to compute its own rate
Distributed Simplex

• Original Simplex method consists of three basic steps:
  1. Column choice
  2. Row choice
  3. Pivot

• A straightforward parallelization scheme within the standard method involves dividing up the columns amongst processors.

• We would now have five steps:
  1. Column choice: each processor will price out its columns and choose a locally best column
  2. Processors will communicate for the local best columns. All that is sent is the pricing value of the processors best column. Afterwards, each processor will know the winner processor who has the global column choice.
  3. Row choice by the winning column.
  4. Broadcast of the winning processors winning column and choice of row.
  5. Simultaneous pivot by all processors on their columns.

mpoutsis@aueb.gr
Example

- Component 1
- Component 2
- Component 3
- Component 4

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>S4</th>
<th>S5</th>
<th>S6</th>
<th>S7</th>
<th>S8</th>
<th>S9</th>
<th>S10</th>
<th>S11</th>
<th>S12</th>
<th>S13</th>
<th>S14</th>
<th>S15</th>
<th>S16</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>16.5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>0.0074</td>
<td>0.016</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0.006</td>
<td>0.018</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-10</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>120</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>-10^-3</td>
<td>-10</td>
<td>-10^-3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

mpoutsis@aueb.gr
Execution Phases

• **Preparation**: Burst management Components communicate through *multicast* to exchange:
  – Ci values (might be known)
  – Application’s Deadlines (might be known)
  – CPuci (might be known)
  – Priority values
  – Minimum and Maximum Rates

• Every component should be able to develop the tableu but it is only interested to keep *columns*:
  – that refer to its own rates
  – simplex slack variable’s columns that are being shared *linearly*
  – and the Tableu’s *last column* (to reduce communication issues)
Execution Phases(2)

- **Computation and Communication**: This phase is used to identify the optimal solutions

- Burst Management Components will run the D.S. algorithm

- If a feasible solution wasn’t found:
  1. We set the low priority components minimum rate at half and re-run D.S.
  2. If we haven’t found a feasible solution we set their Min rates to 1.
  3. Again, if there is no feasible solution we continue with steps 1, 2 to higher priority components

- After having found the optimal solution, iterations will stop and the Burst Management Component at each component will define the solution for its own rate
Execution Phases (3)

- **Rate adaptation – Online**
  - Components periodically check their computation times.
    - If they diverge +/- 10% compared to previous measurements, they adjust their rates accordingly.
    - When execution time for rq has been increased >10% compared to the Deadline, the component notify all applications and D.S. re-runs.
      To have such a situation, new components have emerged since D.S. had run, so their rates haven’t been calculated.

- **When new applications arrive they start at their minimum rate.**
  - If the system can handle their rates, some other components might have to lower their own rates but we don’t have to call the Simplex method.
  - If the system can’t handle their rates, components will notify the others to run the D.S. method and re-adjust the rates.
Experimental Setup

- Tested On CS Lab PCs:
  - Intel Pentium 4, 3.00GHz - 1.5GB RAM - Windows XP
- Code Written in JAVA 6u26
  - More than 2500LOC – also used 6 external libraries

- Used Datasets:
  - Synthetic dataset: ADUs that range from few Kbytes up to 1MByte
  - Mobile Century Dataset from UC Berkeley: Geographical Coordinates

- Application 1: Encryption and Decryption Components
- Application 2: Measuring Distances from specific geographical spots

- Our technique is being also compared with Load Shedding

mpoutsis@aueb.gr
Experiments

Rate over #ADU(rq) using ARADRTS

- Always between rate QoS demands
  - app1: 20-30
  - app2: 10-15

- DS wasn’t emerged

Rate over #ADU(rq) using Load Shedding

- Lots of ADUs(rq) don’t meet rate QoS demands
  - app1: 20-30
  - app2: 10-15

- After application 2 arrived only 35% of the ADUs were between rate QoS demands

mpoutsis@aueb.gr
Experiments

**e2e times over #ADU(rq) for application 1**

- Load Shedding manages better times but both techniques stay under Deadline(rq) = 2000

**e2e times over #ADU(rq) for application 2**

- Deadline(rq) = 2500
Experiments

CPU usage over time

• ARADRTS used avgCPU=17%
• Load Shedding used avgCPU=22%

mpoutsis@aueb.gr
Experiments

Throughput for both techniques

mpoutsis@aueb.gr
Experiments

Rate over #ADU(rq) using ARADRTS

• Always between rate QoS demands
  app1: 20-40
  app2: 10-15

• DS emerged a little after application 2 arrived

Rate over #ADU(rq) using Load Shedding

• Lots of ADUs(rq) don’t meet QoS rate demands
  app1: 20-40
  app2: 10-15

• After application 2 arrived only 47% of the ADUs were between rate QoS demands

mpoutsis@aueb.gr
Experiments

- **Application 1**
  - **Deadline(rq) = 2000**

- **Application 2**
  - **Deadline(rq) = 2500**

---

**e2e times over #ADU(rq) for application1**
- **Deadline(rq) = 2000**

**e2e times over #ADU(rq) for application2**
- **Deadline(rq) = 2500**

mpoutsis@aueb.gr
Experiments

- ARADRTS used avgCPU=14%
- Load Shedding used avgCPU=20%
- Obvious difference after application2 arrived (t=100sec)

mpoutsis@aueb.gr
Experiments

Throughput for both techniques

mpoutsis@aueb.gr
Related Work

• Brown University: **Dynamic load distribution in the Borealis stream processor**, ICDE (Centralized approach)

• Georgia Institute of Technology: **Resource-aware distributed stream management using dynamic overlays**, ICDCS 2005 (Clustered architecture)

• IBM: **Online optimization for latency assignment in distributed real-time systems**, ICDCS 2008 (Lagrange Multipliers)

• IBM, University of Massachusetts: **Distributed resource management and admission control of stream processing systems with max utility**, ICDCS 2007 (Lagrange Multipliers)

• University of California: **BARRE - Accommodating bursts in distributed stream processing systems**, IPDPS 2009 (Offline phase)

• ETH Zurich, Brown University: **Staying FIT: Efficient load shedding techniques for distributed stream processing**, International Conference on Very Large Data Bases 2007 (Uses load shedding)

mpoutsis@aueb.gr
Conclusions

• We presented an algorithm for adaptive rate allocation within the DSPS to meet the end-to-end execution time and rate demands of the applications

• Our approach:
  – determines the rates of the application components at runtime, with respect to the QoS constraints, to compensate for delays experienced by the components or to react to sudden bursts of load
  – It is distributed and low-cost
  – Experimental results illustrate that it is practical, depicts good performance and has low resource overhead

mpoutsis@aueb.gr
Thank you

• Questions?

mpoutsis@aueb.gr