TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Addressing the Challenges of Data Stream Processing

Corso di Sistemi e Architetture per Big Data

A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

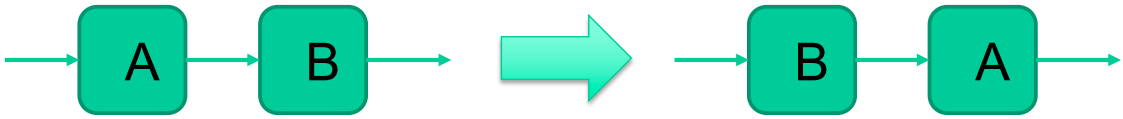
Challenges

- We consider how to tackle some key challenges in DSP systems

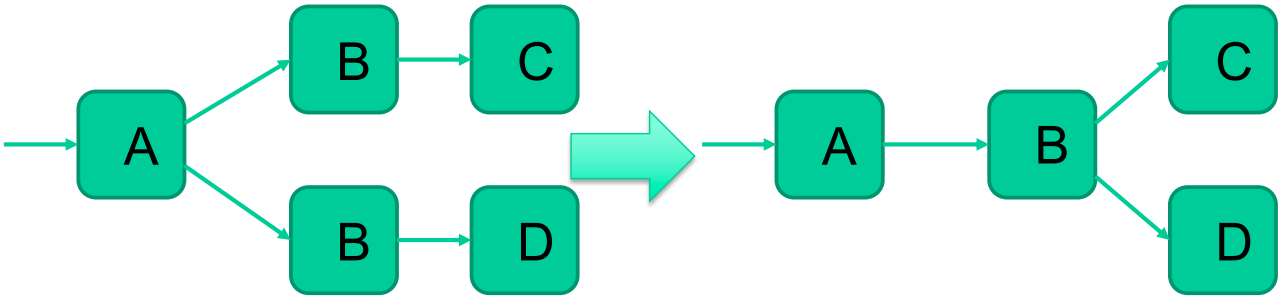
1. Optimizing DSP application
2. Placing DSP operators on computing infrastructure
3. Managing load variations
4. Self-adaptation at runtime
5. Managing stateful operators
6. Fault tolerance

Challenge 1: Optimizing DSP application

- Apply optimizations to the streaming graph
  - At design time (most common) or at runtime
- Operator reordering
  - Avoid unnecessary data transfers

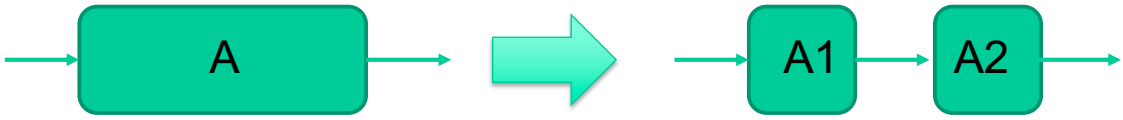


- Redundancy elimination

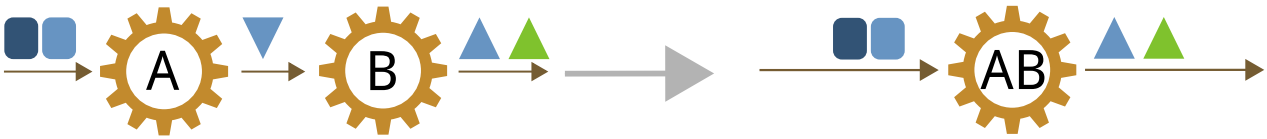


Challenge 1: Optimizing DSP application

- Operator separation

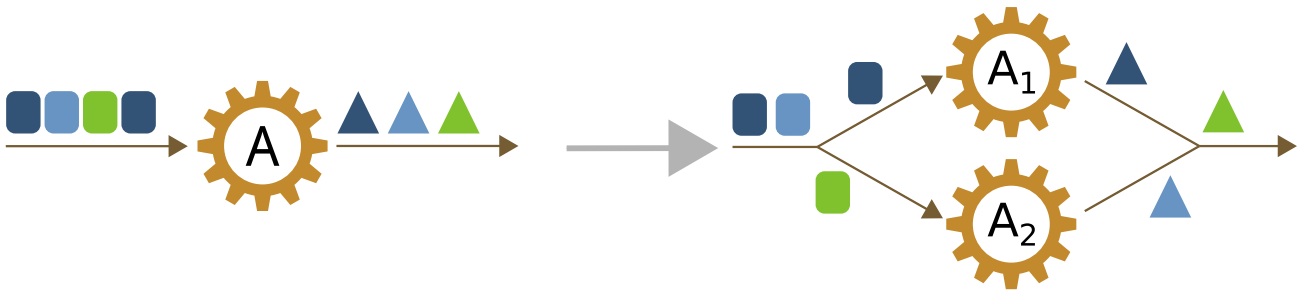


- Operator fusion



Challenge 1: Optimizing DSP application

- Operator scaling (aka *operator fission*)

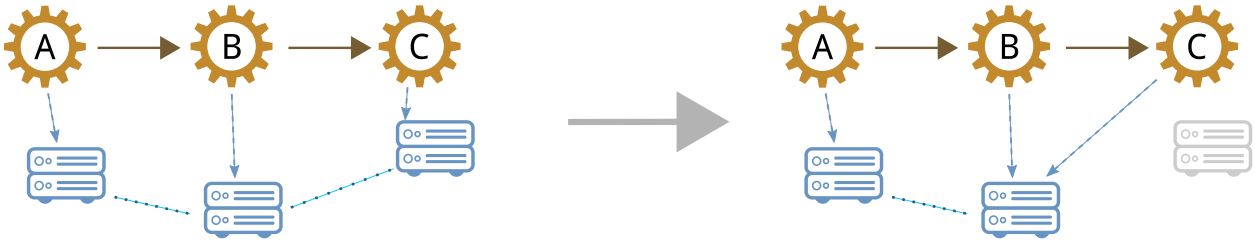


At the streaming system layer

- DSP applications are typically optimized at the DSP application layer and design time
- Can optimization be performed at the *streaming system layer*?
- Can it be *adapted at runtime*?
- We consider two approaches for improving performance (e.g., controlling application latency) at the streaming system layer
  - **Placing DSP operators**
  - **Managing load variations**

Challenge 2: Placing DSP 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: Placing DSP operators

- Placement: a complex problem
  - Trade communication cost against resource utilization
- Challenges to tackle, especially in the Edge-Cloud continuum
  - Non-negligible network latencies
    - E.g., geo-distributed resources
  - Heterogeneity in computing and networking resources
    - E.g., capacity limits , business constraints
  - Computing/network resources can be unavailable
  - Computational requirements of DSP applications may be unknown a-priori and change at runtime
  - DSP applications are long-running

➡️ Need to adapt to internal and external changes

Challenge 2: Design alternatives

- **When** to place operators
  - Initial (static) operator placement
    - Can be more expensive and comprehensive
  - Can also be at runtime
    - Place again all the operators or only a subset

- **How** to determine the placement
  - Mathematical programming
    - Optimal operator placement: NP-hard problem
    - Does not scale well, but provides a benchmark
  - Heuristics
    - Majority of policies
  - Deep Reinforcement Learning

Placement: Design alternatives

- **Who** is the decision maker?
  - Centralized placement strategies
    - Require global view (full resource and network state, application state, workload information)
    - ✓ Capable of determining optimal global solution
    - ✗ Scalability
  - Decentralized placement strategies
    - Take decision based only on local information
    - ✓ Scalability, better suited for runtime adaptation
    - ✗ Optimality is not guaranteed

ODP: Optimal DSP Placement

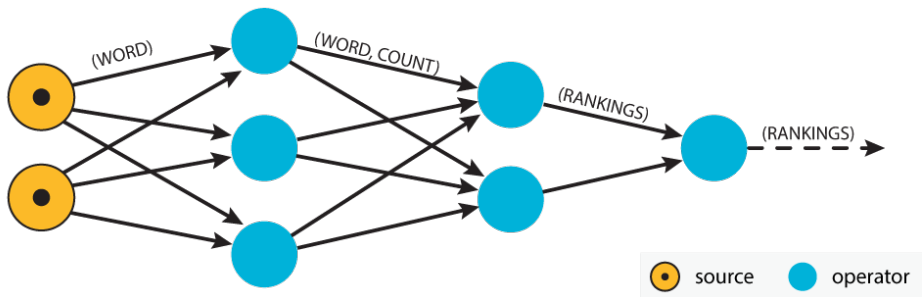
- We proposed **ODP** policy
  - Centralized policy for optimal placement of DSP applications
  - Formulated as Integer Linear Programming (ILP) problem

- Our goals:
  - Compute optimal placement
  - Provide a unified general formulation for DSP application placement (but not only DSP!)
  - Consider multiple QoS attributes of applications and resources
  - Provide a benchmark for heuristics

V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal Operator Placement for Distributed Stream Processing Applications, DEBS '16
http://www.ce.uniroma2.it/publications/PER2016.pdf

ODP placement policy: model

DSP application



Operators

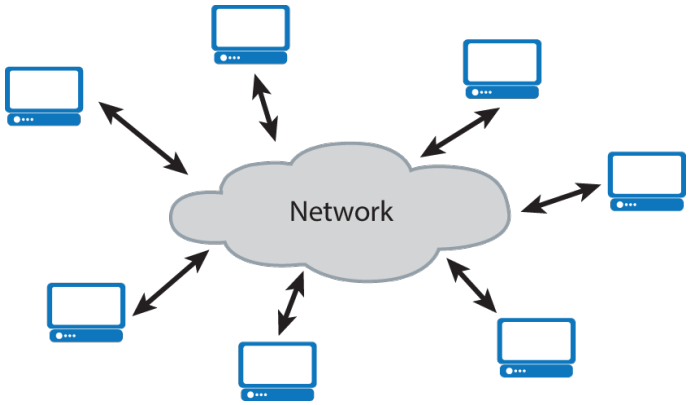
- C_i: required computing resources
- R_i: execution time per data unit

Data streams

- $\lambda_{i,j}$: data rate from operator i to j

ODP placement policy: model

Computing and network resources



Computing resources

- C_u: amount of resources
- S_u: processing speed
- A_u: resource availability

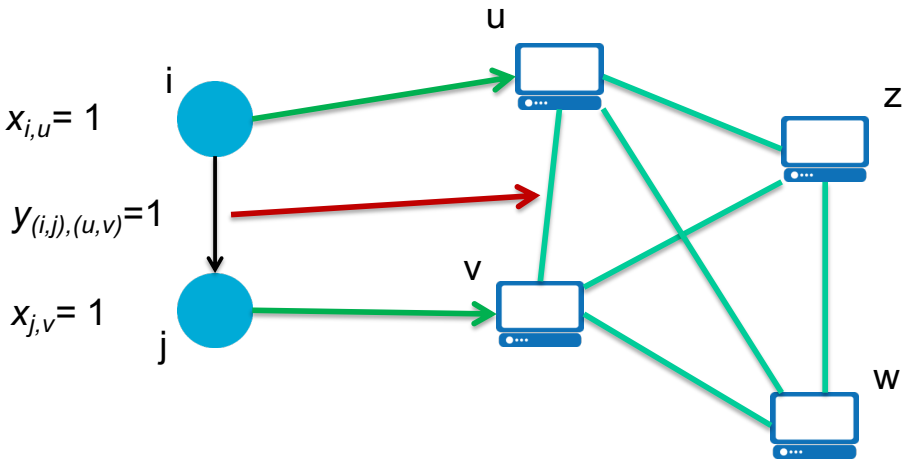
(Logical) Network links

- $d_{u,v}$: network delay from u to v
- $B_{u,v}$: bandwidth from u to v
- $A_{u,v}$: link availability

ODP placement policy: model

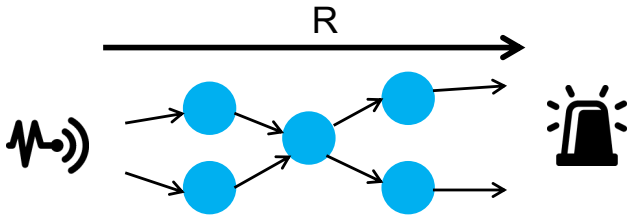
Decision variables

- Determine where to map DSP operators and data streams



ODP placement policy: QoS metrics

- **Response time**
  max end-to-end delay between sources and destination



- **Application availability**
  probability that all resources are up and running
- **Inter-node traffic**
  overall network data rate
- **Network usage**
  in-flight bytes

$\Sigma_{links \in I}$ rate(I)Lat(I)

ODP placement policy: formulation

Tunable knobs to set the optimal placement goals

$$\max_{\boldsymbol{x},\boldsymbol{y},r} F(\boldsymbol{x},\boldsymbol{y},r)$$

subject to:

Latency

$$r \geq \sum_i \sum_u \frac{R_i}{S_u} x_{i,u} + \sum_{(i,j)} \sum_{(u,v)} d_{(u,v)} y_{(i,j),(u,v)} \qquad \forall p \in \pi_G$$

Availability

$$a(\boldsymbol{x},\boldsymbol{y}) = \sum_i \sum_u a_u x_{i,u} + \sum_{(i,j)} \sum_{(u,v)} a_{(u,v)} y_{(i,j),(u,v)}$$

Network bandwidth and node capacity constraints

$$B_{(u,v)} \geq \sum_{(i,j)} \lambda_{(i,j)} y_{(i,j),(u,v)} \qquad \forall u \in V_{res}, v \in V_{res}$$

$$\sum_i C_i x_{i,u} \leq C_u \qquad \forall u \in V_{res}$$

Assignment and integer constraints

$$\sum_u x_{i,u} = 1 \qquad \forall i \in V_{dsp}$$

$$x_{i,u} = \sum_v y_{(i,j),(u,v)} \qquad \forall (i,j) \in E_{dsp}, u \in V_{res}$$

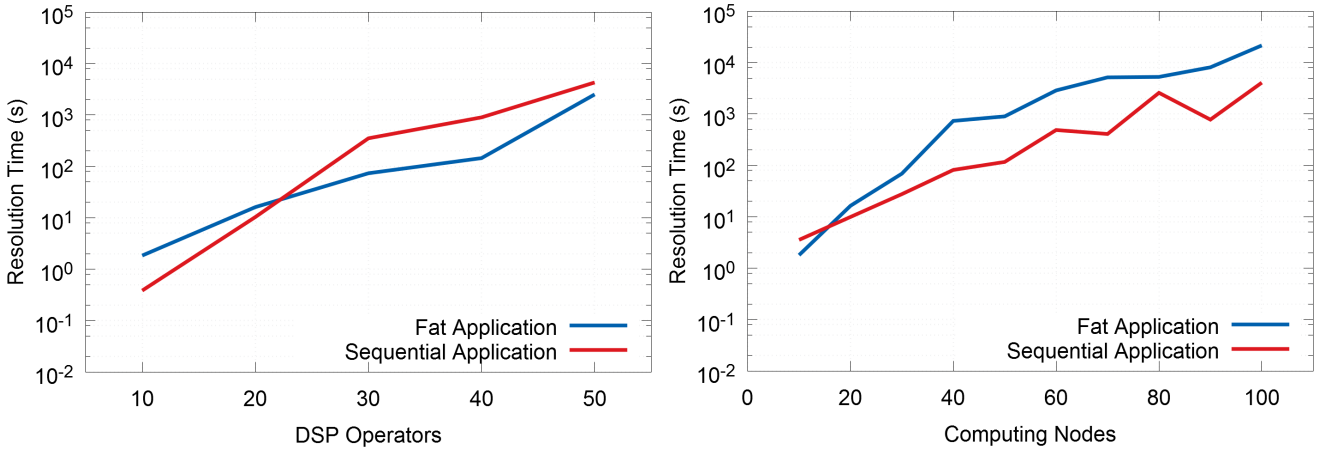
$$x_{j,v} = \sum_u y_{(i,j),(u,v)} \qquad \forall (i,j) \in E_{dsp}, v \in V_{res}$$

$$x_{i,u} \in \{0,1\} \qquad \forall i \in V_{dsp}, u \in V_{res}$$

$$y_{(i,j),(u,v)} \in \{0,1\} \qquad \forall (i,j) \in E_{dsp}, (u,v) \in E_{res}$$

ODP placement policy: scalability issue

- Optimal placement is NP-hard: does not scale with system/application size



- We need heuristics to compute placement in a feasible amount of time

Centralized placement heuristics

- Idea: reduce inter-node communication and balance CPU load by co-locating communicating tasks
- Approach: use a centralized greedy heuristic to guide placement

Greedy heuristic steps:

1. Rank operator pairs according to exchanged traffic
2. For each operator pair:
   - If neither operator has been assigned yet, then assign both to the same node
   - Otherwise, evaluate the node where the assigned operator is placed and the least loaded node and choose the configuration that minimizes inter-process communication

L. Aniello, R. Baldoni and L. Querzoni, Adaptive online scheduling in Storm, DEBS '13
http://midlab.diag.uniroma1.it/articoli/ABQ13storm.pdf

Decentralized placement heuristic

- Heuristics goal: reduce network usage
  - Network usage metric combines link latencies and exchanged data rates among DSP operators:

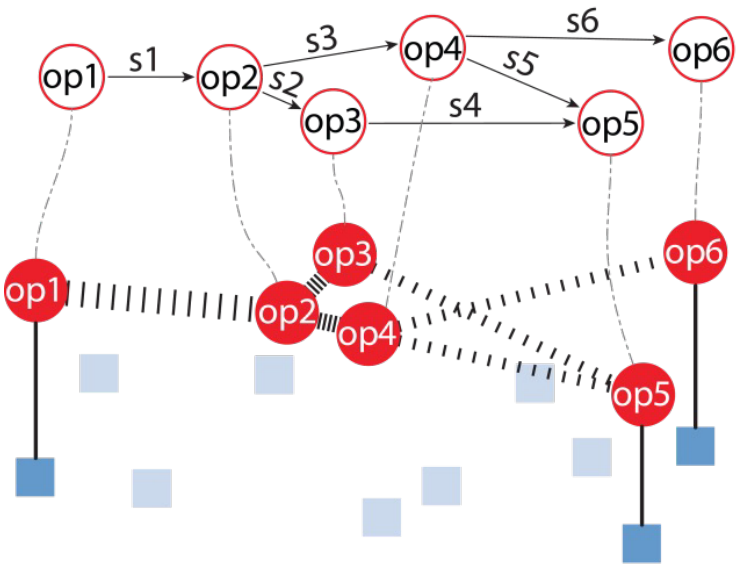
$$\sum_{links \in I} rate(l)Lat(l)$$

- Idea: exploit **spring relaxation**
  - DSP application regarded as a system of springs, whose minimum energy configuration corresponds to minimizing network usage
- Features
  - Decentralized policy to minimize network impact
  - Adaptive to change in network conditions

P. Pietzuch et al., Network-aware operator placement for stream-processing systems, ICDE '06 https://www.doc.ic.ac.uk/~prp/manager/doc/icde06-camera-ready.pdf

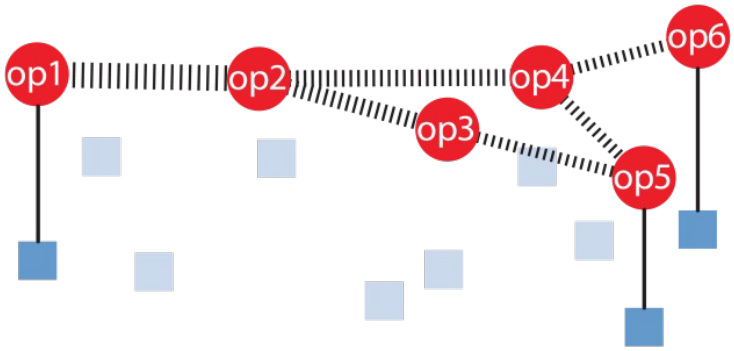
Decentralized placement heuristic

1. Represent DSP application as an equivalent system of springs



Decentralized placement heuristic

2. Determine operator placement in the cost space by minimizing the elastic energy of the equivalent system

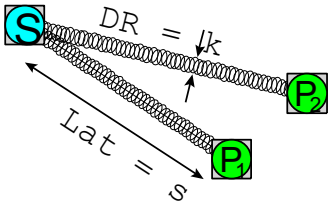


Network of springs tries to minimize potential energy E

$$E = \sum_{l \in L} DR(l) Lat(l)^2$$

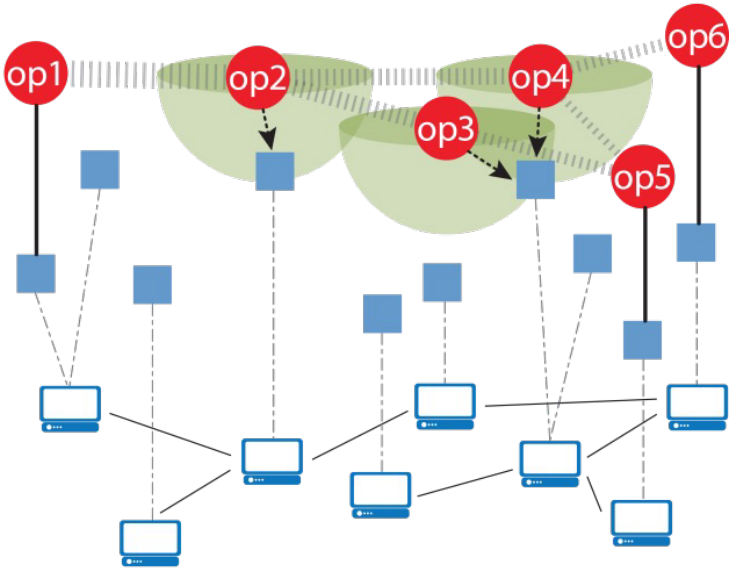
Streams as springs, that restore a force F = ½ • k • s:

- k (spring constant): exchanged data rate on link
- s (spring extension): latency on link



Decentralized placement heuristic

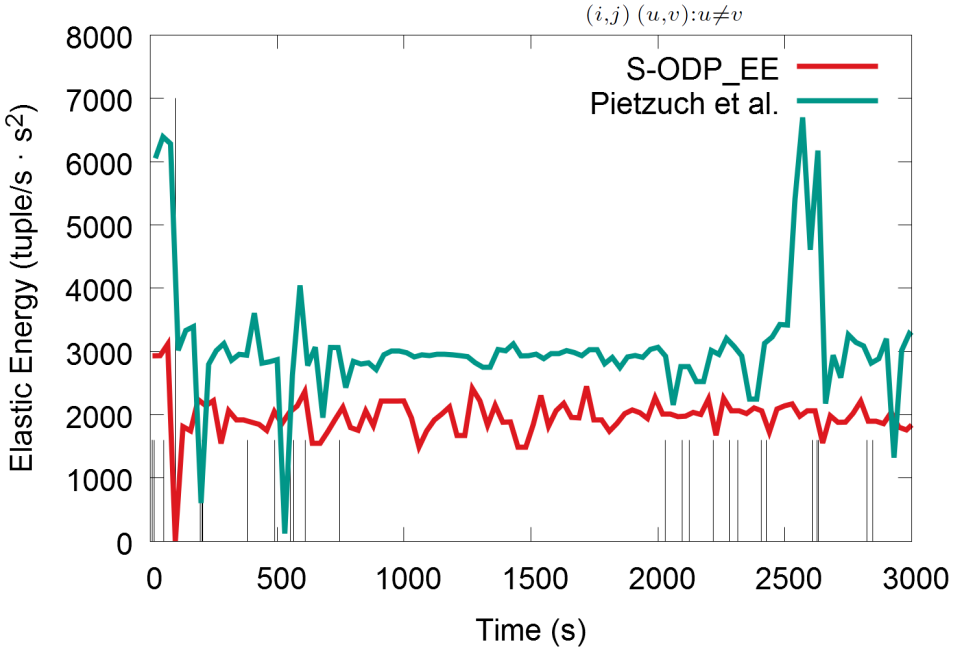
3. Map decision onto physical nodes



ODP as benchmark

Distributed placement heuristic that minimizes network usage

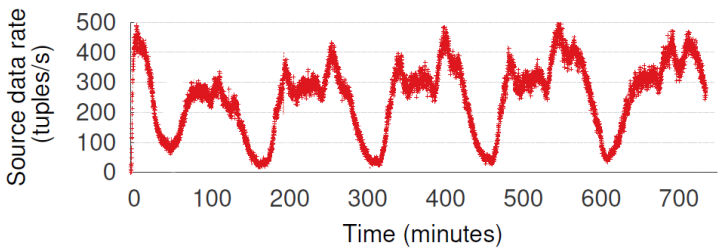
Pietzuch et al. : $\min EE(\boldsymbol{y}) = \min \sum_{(i,j)} \sum_{(u,v):u \neq v} \lambda_{(i,j)} d^2_{(u,v)} y_{(i,j),(u,v)}$



Optimal Operator Placement for Distributed Stream Processing Applications, http://www.ce.uniroma2.it/publications/PER2016.pdf

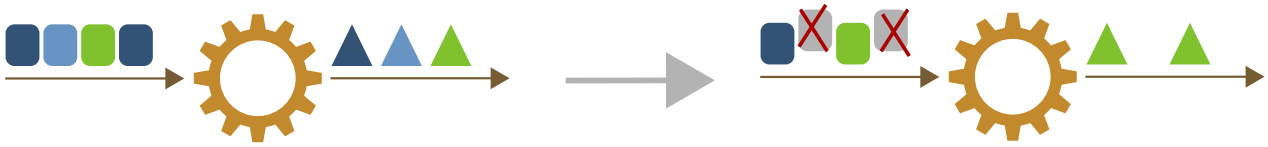
Challenge 3: Manage load variations

- Typical characteristics of stream processing workloads:
  - High data volume and high ingestion rates
    - E.g., millions of posts, comments, likes, and shares are generated every second
  - Bursty behavior with sudden spikes in workload
    - Traffic volume can spike suddenly during major events, like breaking news or emergency situations



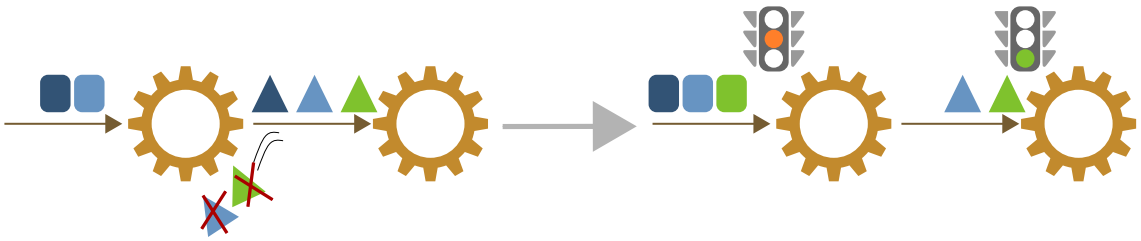
Approaches to manage load variations

1. Admission control
   - Mechanism that decides whether a new data flow can be accepted and processed by the system
2. Static reservation
   - Pre-allocating specific resources (e.g., CPU, memory) in advance
   - *Cons*: may lead to over-provisioning and increased cost
3. Load shedding
   - Dynamic strategy that selectively drops data tuples when system load exceeds a threshold (e.g., high CPU usage)
   - *Cons*: reduces accuracy and completeness of output
   - Needs careful tuning to minimize impact on quality



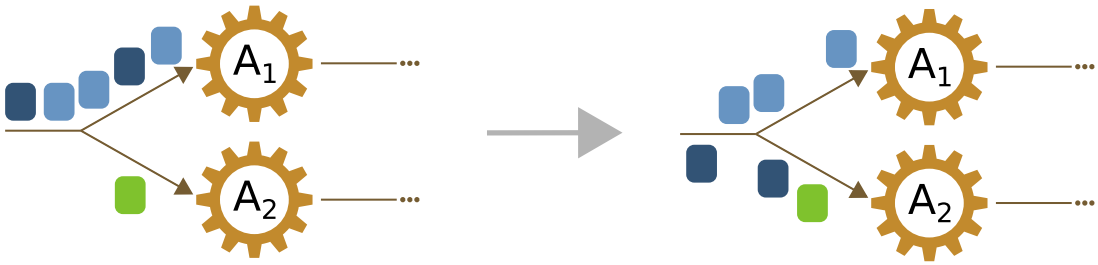
Approaches to manage load variations

4. Backpressure
   - Adaptive rate allocation mechanism to handle bottlenecks in streaming pipelines
   - The upstream operator before a bottleneck stores incoming data in an *internal buffer* to slow down data flow
   - Can propagate recursively upstream, all the way back to the data sources



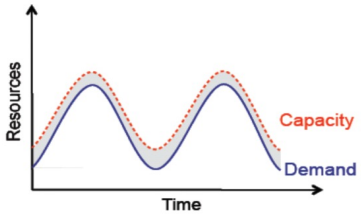
Approaches to manage load variations

5. Redistribute load, that is adjusting the system to balance workload, for example by:
   - Changing stream partitioning
   - Determining new operator placements
   - Migrating operators across computing nodes
   - *Cons*: available resources may be insufficient, limiting the effectiveness of redistribution



Approaches to manage load variations: elasticity

- Common approach to handle load variations in distributed applications
  - Detect bottlenecks (in our case, DSP operators)
  - Resolve them through **elasticity**: dynamically acquire or release resources as needed



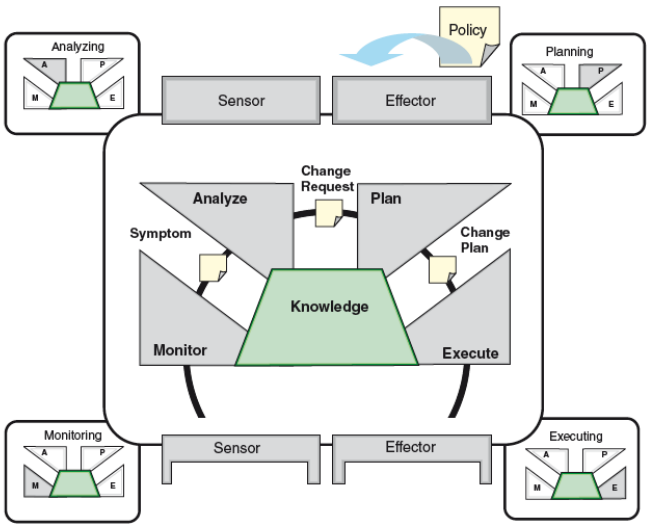
- How?
  - Manually: feasible, but inefficient and error-prone
- Better approach?
  - Enable self-adaptation and adapt application deployment at runtime using the **MAPE** loop

Challenge 4: Self-adapt at runtime

- Many factors may change at runtime, such as:
  - Load variations
  - QoS of computing resources
  - Cost fluctuations due to dynamic pricing
  - Network characteristics
  - Node mobility
- How to adapt DSP application deployment when changes occur?
- Solution: enhance DSP systems with runtime adaptation capabilities
- Possible **adaptation action**
  - Scale-out/in the number of operator replicas
  - Migrate operators across different computing nodes

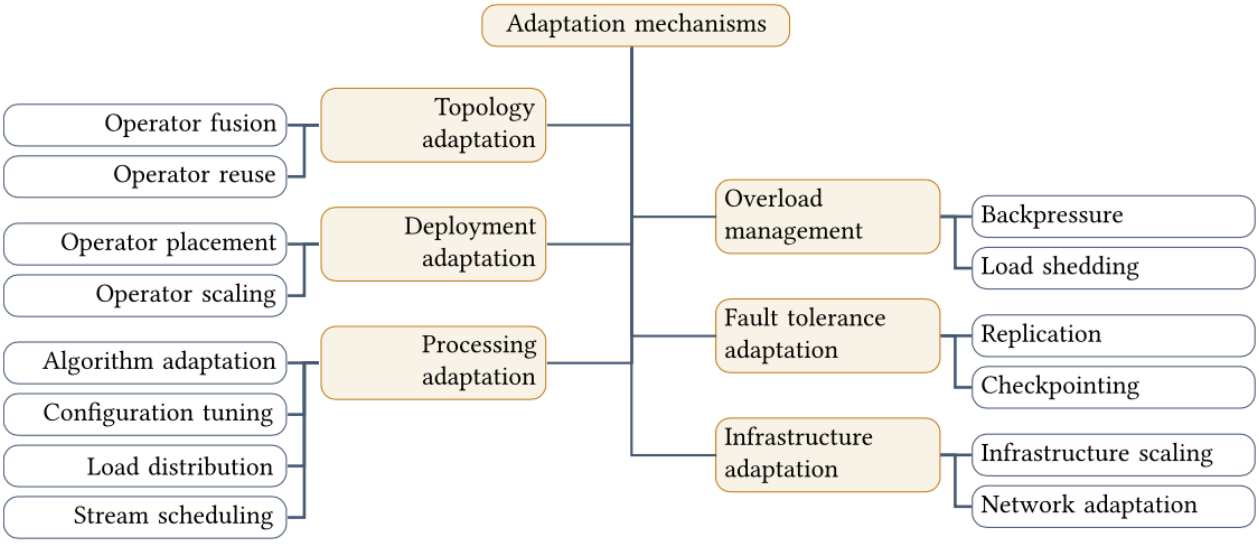
Self-adaptive deployment using MAPE

- **MAPE** (**M**onitor, **A**nalyze, **P**lan and **E**xecute)

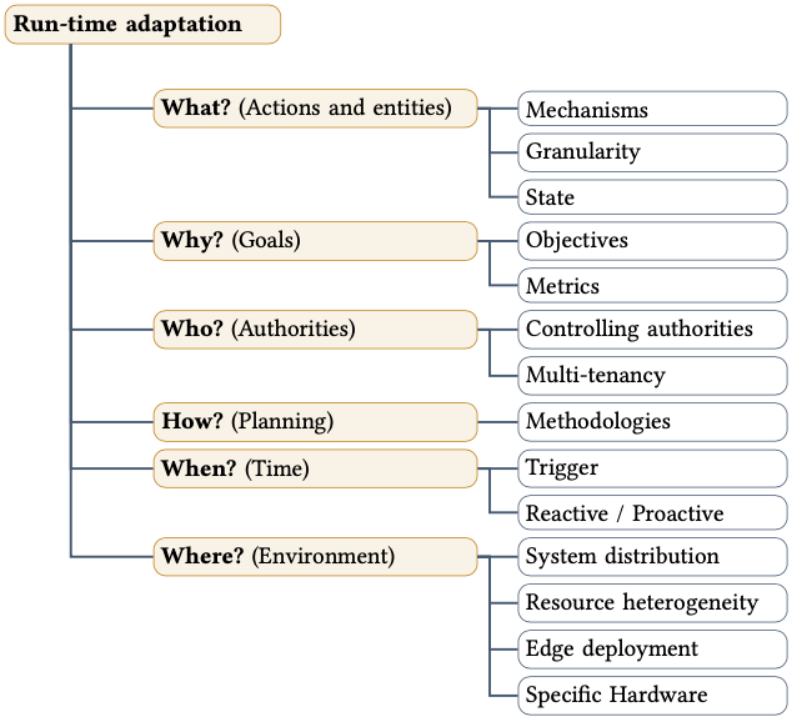


- **Plan** phase: decide how to **adapt** DSP application deployment

Adaptation mechanisms for DSP

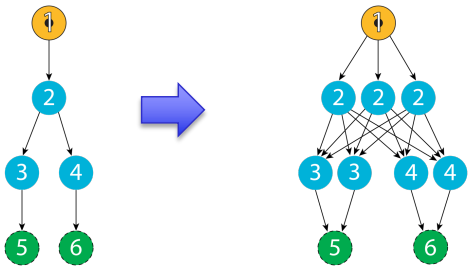


Classifying adaptation solutions for DSP



Scaling data stream processing

- *At which layer*?
  - Application layer (i.e., operator scaling)
    - i.e., apply **SPMD** paradigm: concurrent execution of multiple replicas of the same operator on different partitions of data stream
    - Scale out/in operators by adding/removing operator replicas



  - Infrastructure layer
    - Scale horizontally computing resources (containers, virtual machines, physical machines)
    - Also scale vertically computing resources (containers, virtual machines)

Scaling data stream processing

- *When* and *how* to scale?
  - Open issue, a simple example:
    - When: threshold-based (like AWS Auto Scaling)
    - How: add/remove one operator replica at time
    - Where: determine randomly (or in a round-robin fashion) location of new replica
- Caution: elasticity overhead is not zero
  - Elasticity often requires running new placement decisions to accommodate additional replicas
  - Dynamic scaling can significantly impact stateful operators, introducing overhead in state migration and synchronization

Scaling: limits of centralized approaches

- Centralized optimization algorithms struggle to scale with large problem sizes
- Centralized MAPE architectures face scalability issues in geo-distributed environments
- Although components are distributed, control logic remains centralized, creating bottlenecks
- Solution for Edge-Cloud Continuum: decentralize the MAPE loop to distribute control and improve scalability, i.e., **decentralized MAPE**

How to decentralize MAPE control loop?

- Multiple patterns for decentralized control
  - Each comes with pros and cons
  - Choice depends on system requirements, scalability, and complexity

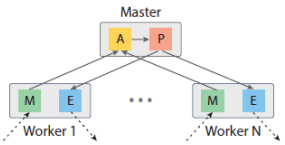


Figure 1: Hierarchical MAPE: master-worker pattern

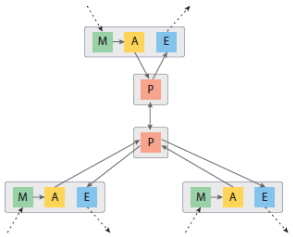


Figure 2: Hierarchical MAPE: regional pattern

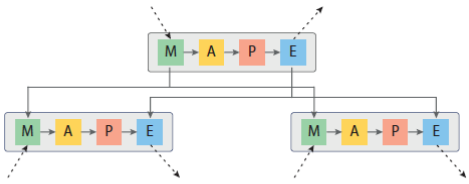


Figure 3: Hierarchical MAPE: hierarchical control pattern

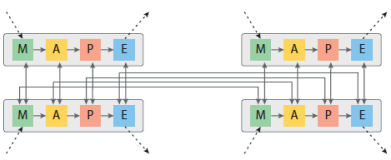


Figure 4: Flat MAPEs: coordinated control pattern

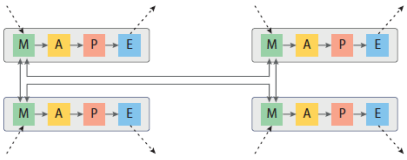
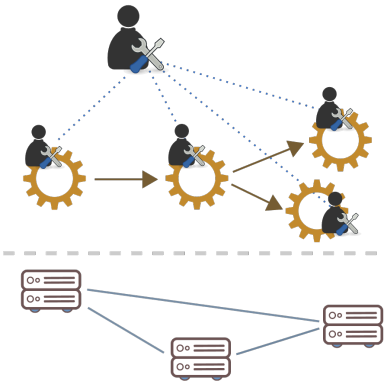


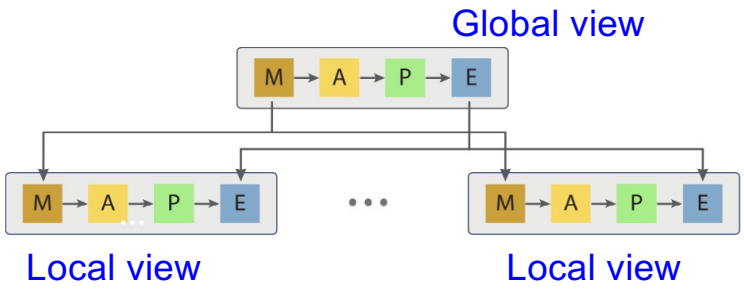
Figure 5: Flat MAPEs: information sharing pattern

D. Weyns et al., On patterns for decentralized control in self-adaptive systems, *SEAMS II*, 2013 https://ics.uci.edu/~malek/publications/2012aSefSAS.pdf

How to decentralize control?

- Our approach:
  - **Hierarchical MAPE** architecture to enable efficient runtime adaptation
  - Distribute MAPE control loops (one global controller and multiple local controllers), balancing global coordination with local autonomy for scalability and responsiveness





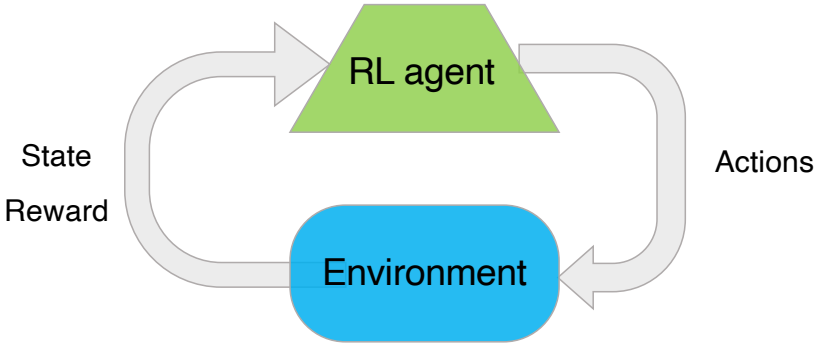
Local elasticity policy

- Let's focus on local Plan policy that controls the elasticity of individual DSP operators
- Make decision with a limited local view
  - e.g., operator's resource utilization and input data rate
- Two classes of elasticity policies
  - Threshold-based policy (e.g., used by AWS Auto Scaling)
    - ✗ Requires manual tuning and domain expertise to choose thresholds
  - Reinforcement Learning-based policies

V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, Decentralized self-adaptation for elastic Data Stream Processing, *Future Generation Computer Systems*, 2018
http://www.ce.uniroma2.it/publications/fgcs2018_dsp.pdf

Reinforcement Learning in a nutshell

- Branch of ML dealing with sequential decision-making
- Agent interacts with environment through actions and receives feedback in the form of reward (paid cost)
- Goal: learn to act as to maximize (minimize) long-term reward (cost)
- Trial-and-error experience

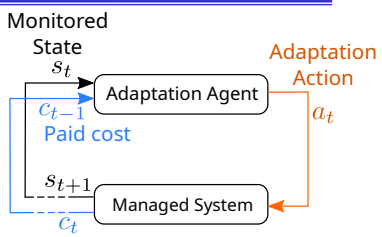


Reinforcement Learning in a nutshell

- We considered different classes of RL algorithms:
  - Baseline **model-free** learning algorithms (e.g., Q-learning)
  - **Model-based** learning algorithms that exploit what is known or can be estimated about system dynamics

Sutton and Barto, Reinforcement Learning: An Introduction, 2020
http://incompleteideas.net/book/RLbook2020.pdf

RL-based local elasticity policy



- At each step the RL agent performs an action, looking at current state s_t
- Chosen action a_t causes payment of immediate cost c_t and transition to a new state s_{t+1}
- To minimize expected long-term (discounted) cost, RL agent estimates $Q(s, a)$
  - Q-function: expected long-run cost of taking action a in state s

Algorithm 1 RL-based Operator Elastic Control Algorithm

1: Initialize the Q functions
2: **loop**
3: choose a scaling action a_i (based on current estimates of Q)
4: observe the next state s_{i+1} and the incurred cost c_i
5: update the $Q(s_i, a_i)$ functions based on the experience
6: **end loop**

RL-based local elasticity policy: Q-learning

- Q-learning: baseline model-free RL algorithm
- Given current state, the agent chooses next action
  1. Either exploiting its knowledge about system (i.e., current estimates of Q-function stored in Q-table) by greedily selecting the action that minimizes the estimated future costs
  2. Or exploring by selecting a random action to improve its knowledge about system
     - We consider ε-greedy action selection method

Q-table

State/Action	a_1	a_2	...
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	...
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	...
...	...		
s_n	$Q(s_n, a_1)$	$Q(s_n, a_2)$	...

- Q-learning: update step of Q-function

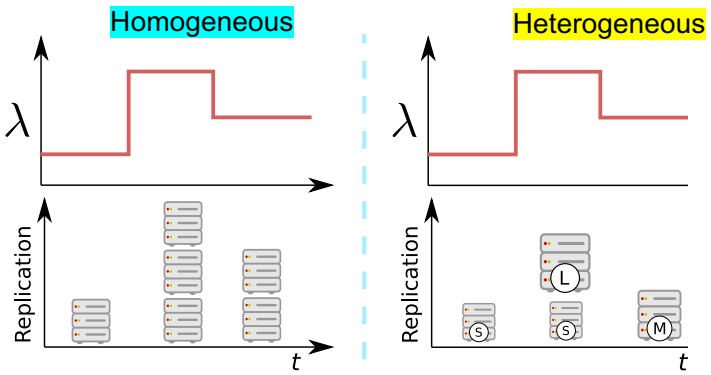
$$Q(s_i, a_i) \leftarrow (1-\alpha)Q(s_i, a_i) + \alpha \left[c_i + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q(s_{i+1}, a') \right]$$

RL-based local elasticity policy: advanced RL techniques

- We have also exploited advanced RL techniques in order to deal with large state space (e.g., due to heterogeneous computing resources)
  - Function Approximation
  - Deep Learning
  - Goal: build approximate representations of state space and achieve near-optimal solutions with reduced memory demand
- We now consider the high-level ideas
  - More details:
    - Our tutorial at Performance 2021: Reinforcement Learning for Run Time Performance Management in the Cloud/Edge https://www.performance2021.deib.polimi.it/www.performance2021.deib.polimi.it/tutorials/
    - Russo Russo et al., Hierarchical Auto-Scaling Policies for Data Stream Processing on Heterogeneous Resources, ACM TAAS, 2023 http://www.ce.uniroma2.it/publications/TAAS2023.pdf

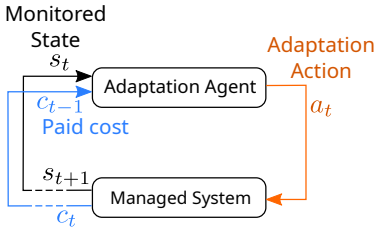
Auto-scaling on heterogeneous nodes: increasing complexity and realism

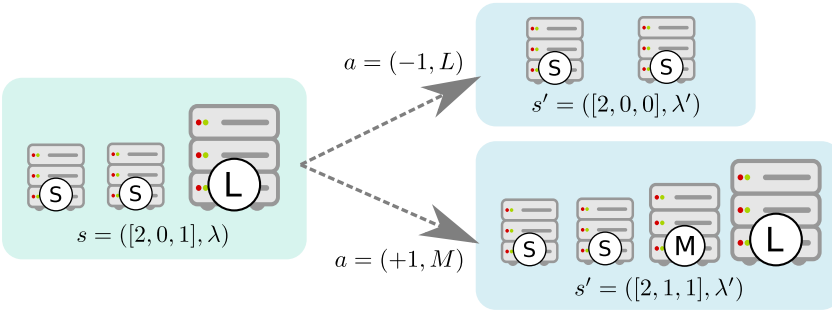
- We consider a heterogeneous computing infrastructure
  - Nodes with different types/amount of resources
- RL agent must decide not only how many replicas to run but also which types of nodes to host them



How to formulate?

- N resource types: T_{res} = {  }
- State s = (**k**, λ)
  - k_i = #replicas on nodes of type i
  - λ = input data rate
- Actions A(s)={(δ,τ): δ∈{−1,+1}, τ ∈T_{res}}∪{do−nothing}
- Cost = w_{res} resource cost + w_{perf} performance + w_{rcf} reconfiguration

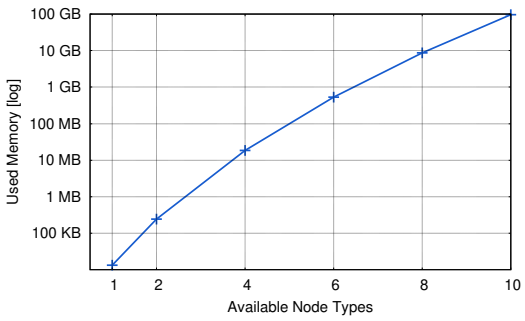


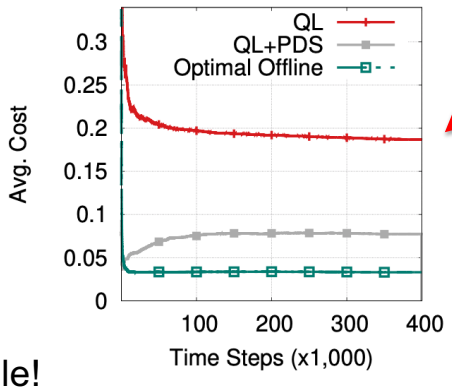


Standard RL algorithm falls short

- Q-learning falls short in heterogeneous DSP context
  - ✗ Too much memory to store tabular representation of Q-function
  - ✗ Very slow convergence

State/Action	a_1	a_2	…
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	…
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	…
…	…		
s_n	$Q(s_n, a_1)$	$Q(s_n, a_2)$	…





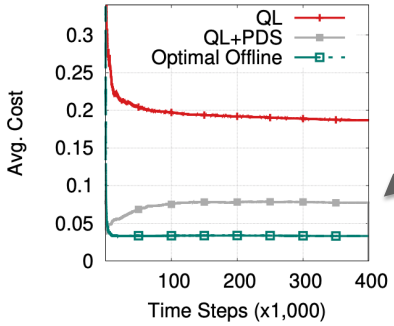
Q-learning

Note: each operator has its own Q-table!

How to improve?

- We exploit multiple solutions

1. Separate the known from the unknown, inject **partial model knowledge** (i.e., post-decision states) and learn only the unknown part
   - Do we really need to learn everything from scratch?
     - We know which is the impact of scaling actions on the current deployment
     - We know whether a reconfiguration cost is paid after a certain action
     - We can estimate performance-related costs through a model

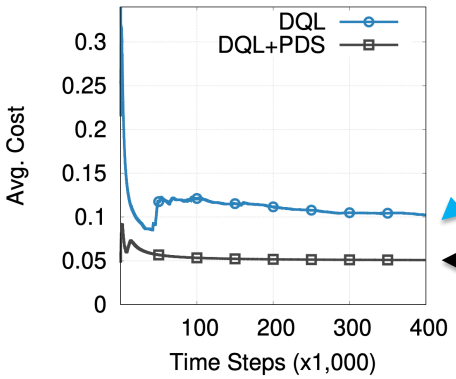


Partial model knowledge

How to improve?

- We exploit multiple solutions

2. Resort to non-linear **function approximation** (deep Q network)
3. Combine all together



Function approximation

Partial model knowledge

+

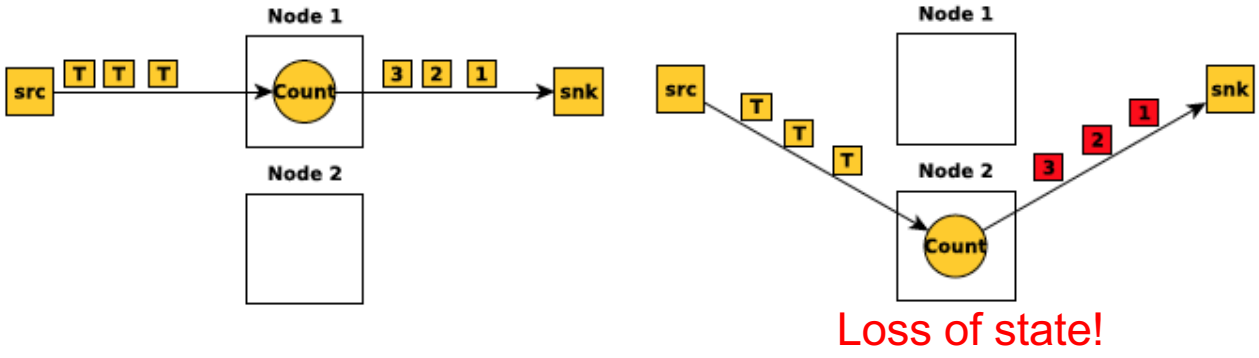
Function approximation

Reconfiguration overhead

- Deployment reconfiguration has a non-negligible cost
  - Can negatively impact application performance in the short term
  - Application freezing times caused by operator migration and scaling, especially with stateful operators
- Solution:
  - Trigger reconfiguration only when needed
  - Take into account reconfiguration overhead into decision-making policy

Challenge 5: Stateful operators

- State complicates things…
  - Dynamic scaling: state must be partitioned, transferred, and rebalanced across operator replicas
  - Operator re-placement: requires **state migration**
  - Recovery from failure: stateful operators require mechanisms such as checkpointing

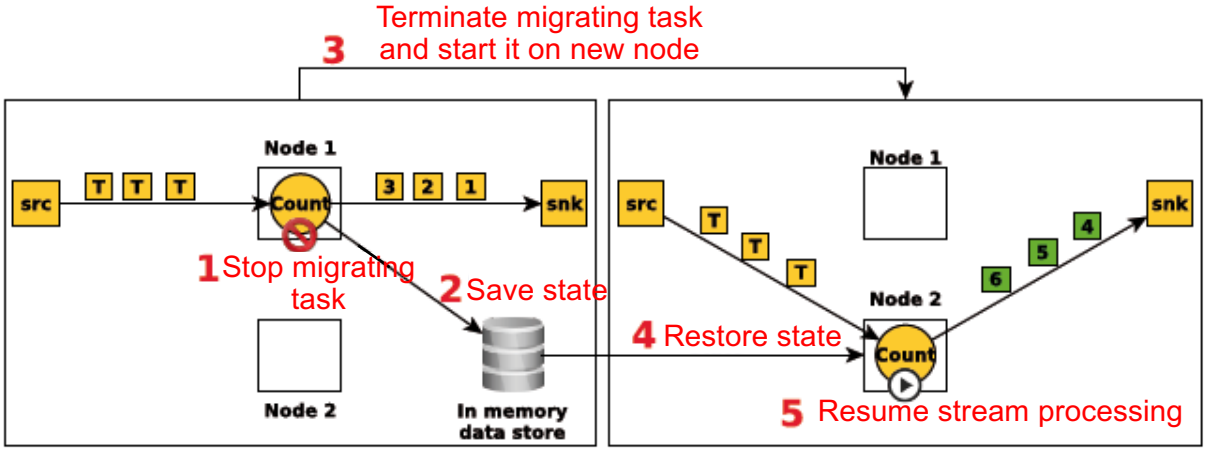


Stateful operator migration

- Support for stateful operator migration varies
  - Some do not support it at all
  - Others do (e.g., research prototypes and production systems like Spark Streaming)
  - Flink: stateful operators can be migrated through savepoints and checkpoints
- Requirements for stateful operator migration
  - Safety
    - Ensure operational consistency during and after migration
    - Prevent data loss or duplicate processing
  - Application transparency
    - Do not require changes to application logic
    - Maintain seamless operation from developer's perspective
  - Low overhead
    - Limited impact on performance and resource usage
    - Fast and efficient state transfer with minimal downtime

Approaches for stateful operator migration

- Main approaches
  1. Pause-and-resume approach
  2. Parallel track approach
- Pause-and-resume
  - ✗ Application latency peak during migration



Stateful operator migration

2. Parallel track
   - Old and new operator instances run concurrently until their state is synchronized
   - ✓ No latency peak
   - ✗ More complex: requires mechanisms for synchronizing the two instances

Stateful operators: other issues

- How to identify which portion of state to migrate? Approaches:
  - Expose an API to let the user manually manage the state
  - Support only 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
- How to balance the load among multiple stateful replicas? Possible approaches:
  - Use consistent hashing
  - Use partial key grouping
    - Use two hash functions where a key can be sent to two different replicas instead of one
  - Only in research prototypes

Challenge 6: Guaranteing fault tolerance

- DSP applications are long-running, making failures inevitable
- Approaches for fault tolerance:
  - Active replication: run multiples copies to ensure availability
  - Checkpointing: periodically save state to recover from failures (e.g., Flink)
  - Replay logs
- Solutions with different trade-offs between runtime cost during normal operation and recovery time
- Large-scale deployments complicate things
  - Network partitions and CAP theorem

References

- M. Hirzel, R. Soulé, S. Schneider, B. Gedik, R. Grimm, A catalog of stream processing optimizations, *ACM Comput. Surv.*, 2014 https://hirzels.com/martin/papers/csur14-streamopt.pdf

- V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo, Run-time adaptation of data stream processing systems: The state of the art, *ACM Comput. Surv.*, 2022 http://www.ce.uniroma2.it/courses/sabd2223/papers/csur2022.pdf