Early Scheduling in Parallel State Machine Replication

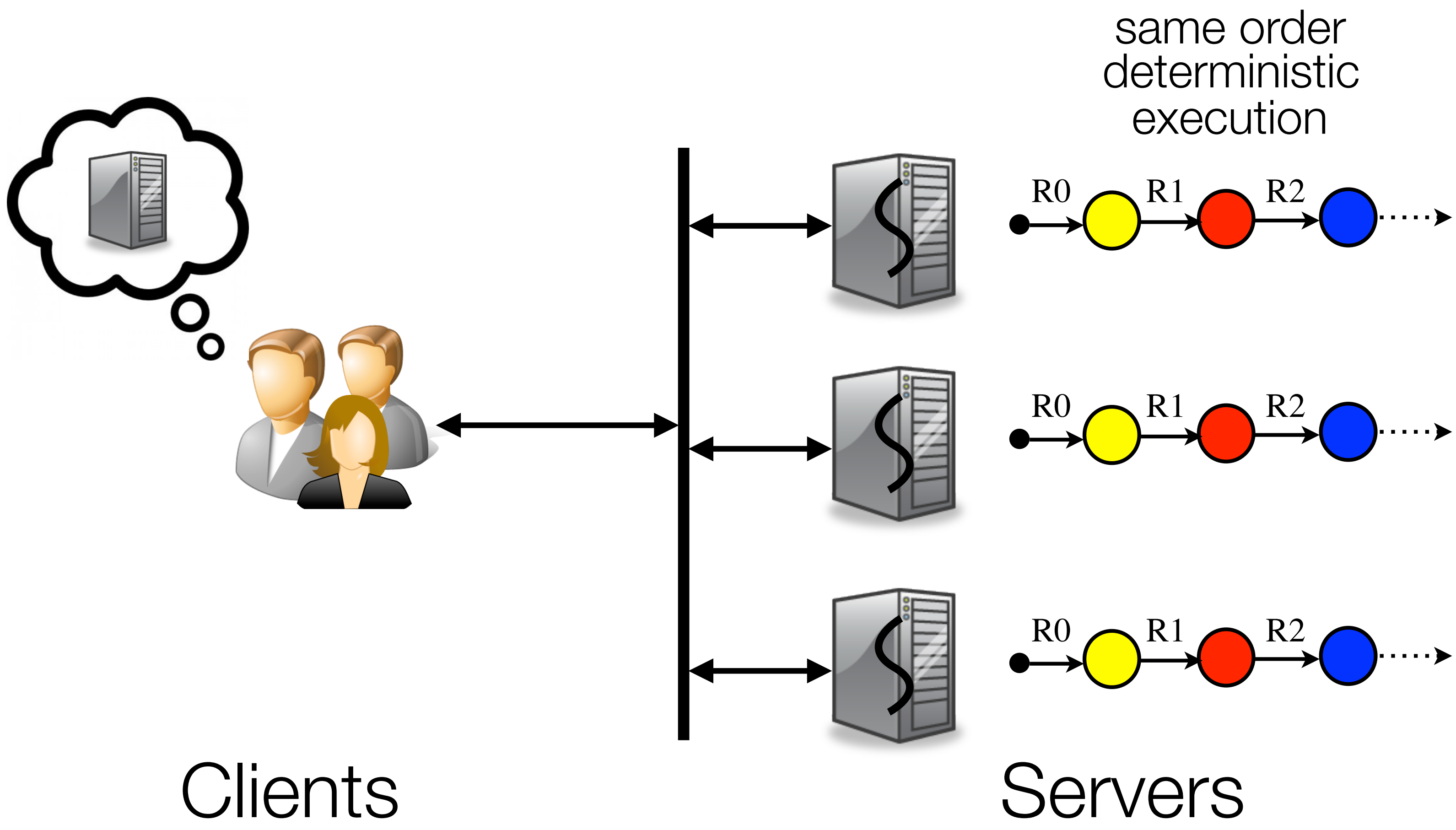
Eduardo Alchieri, Fernando Dotti, and **Fernando Pedone**

Universidade de Brasilia, Pontifica Universidade Católica do Rio Grande do Sul, and **University of Lugano**

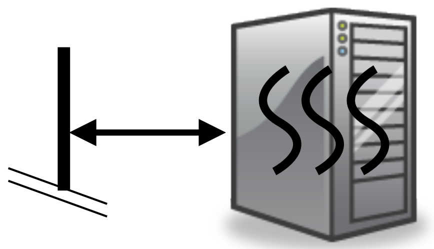
State Machine Replication (SMR)

- Fundamental approach to fault tolerance
  - Google Spanner
  - Apache Zookeeper
  - Windows Azure Storage
  - MySQL Group Replication
  - Galera Cluster
  - Blockchain, …

SMR is intuitive and simple



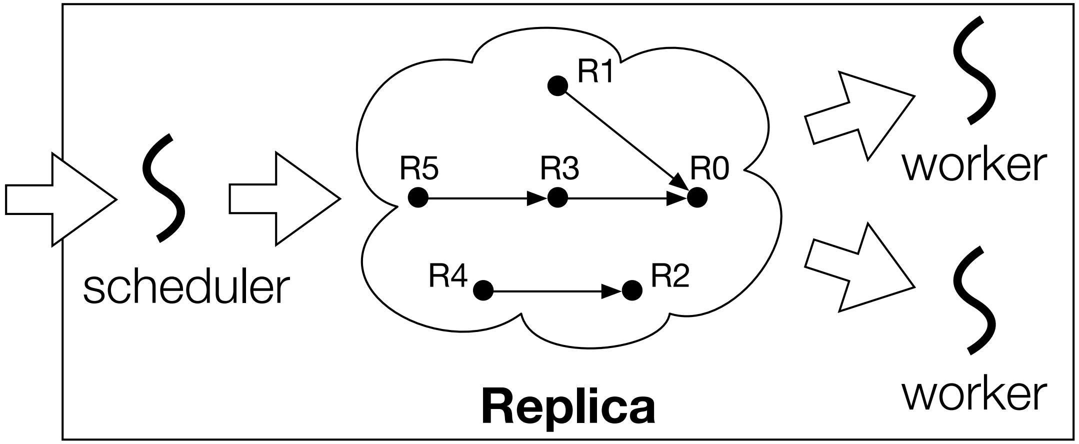
Parallel State Machine Replication

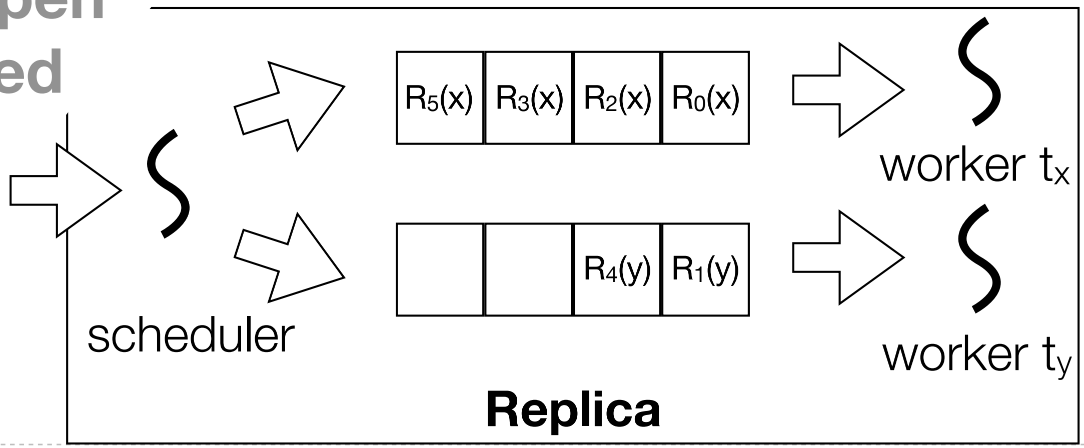


- Key observation
  - Independent requests can execute concurrently
  - Conflicting requests must be serialized and executed in the same order by the replicas
  - Two requests conflict if they access common state and at least one of them updates the state

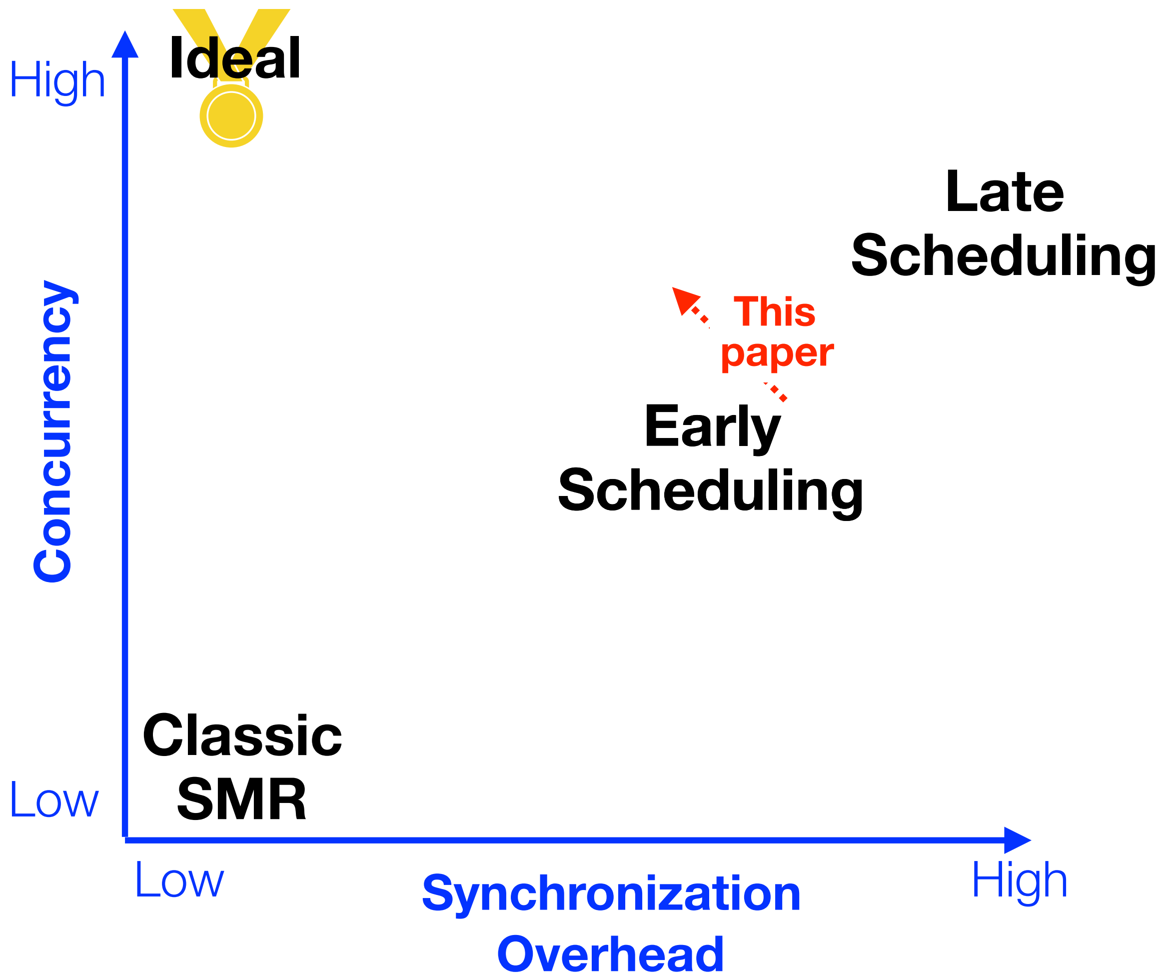
Parallel State Machine Replication

- Late scheduling
  - Scheduling happens after requests are ordered
- Early scheduling
  - Scheduling decisions happen before requests are ordered
  - E.g., worker t_x executes requests on X, worker t_y executes requests on Y





Scheduling tradeoff

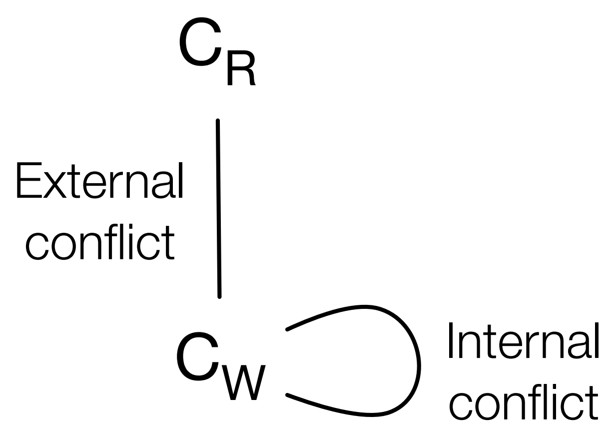


Our contributions

- Generalization of Early Scheduling
    - Classes of requests: expressing application concurrency
    - How to automatically map classes to worker threads
    - How the resulting technique compares to late scheduling

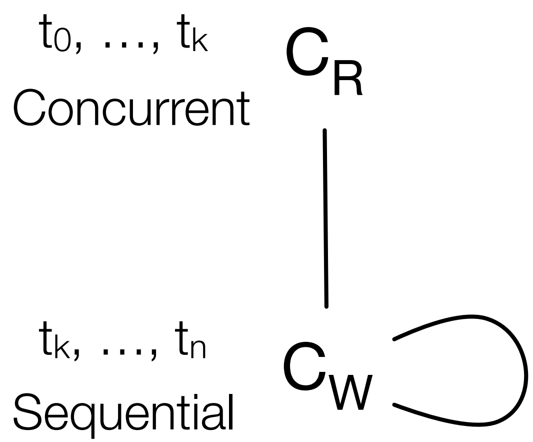
Classes of requests

- Readers and writers
  - Class C_R: read requests
  - Class C_W: write requests

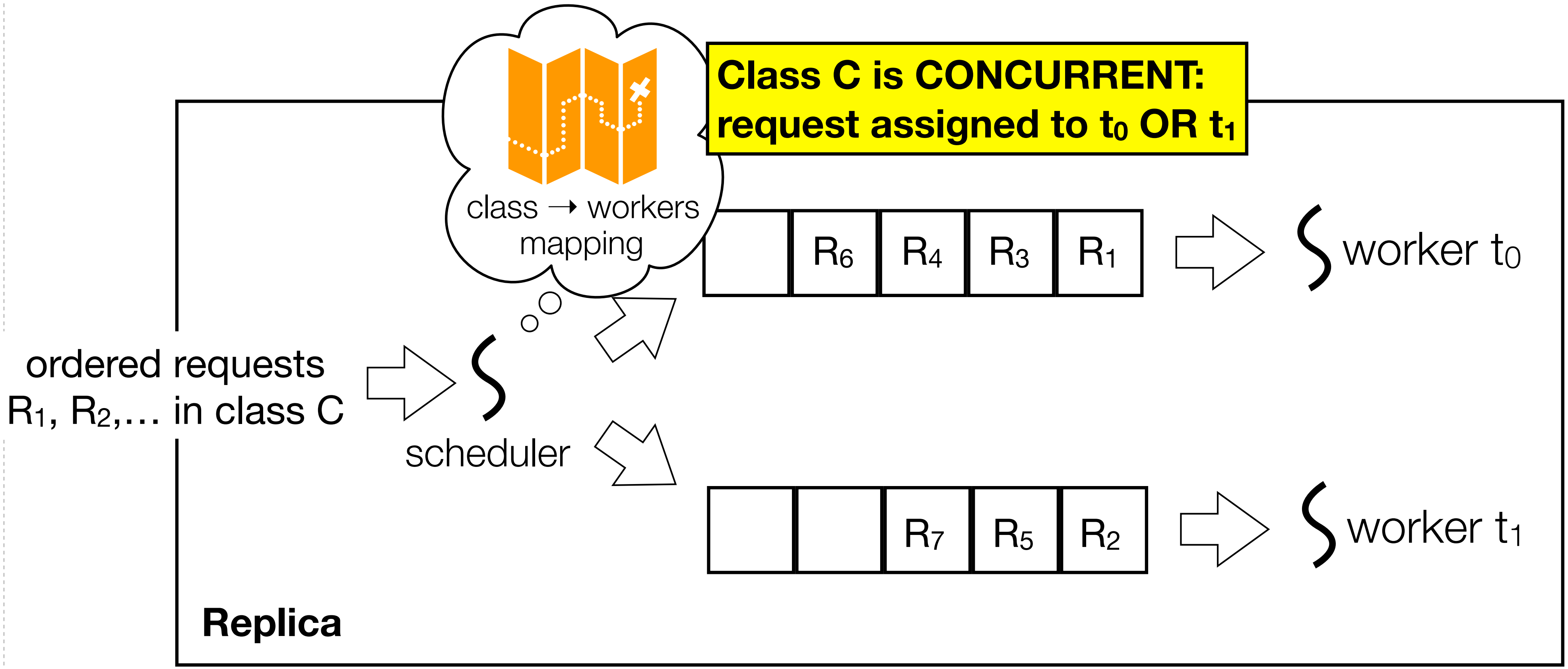


Mapping classes to workers

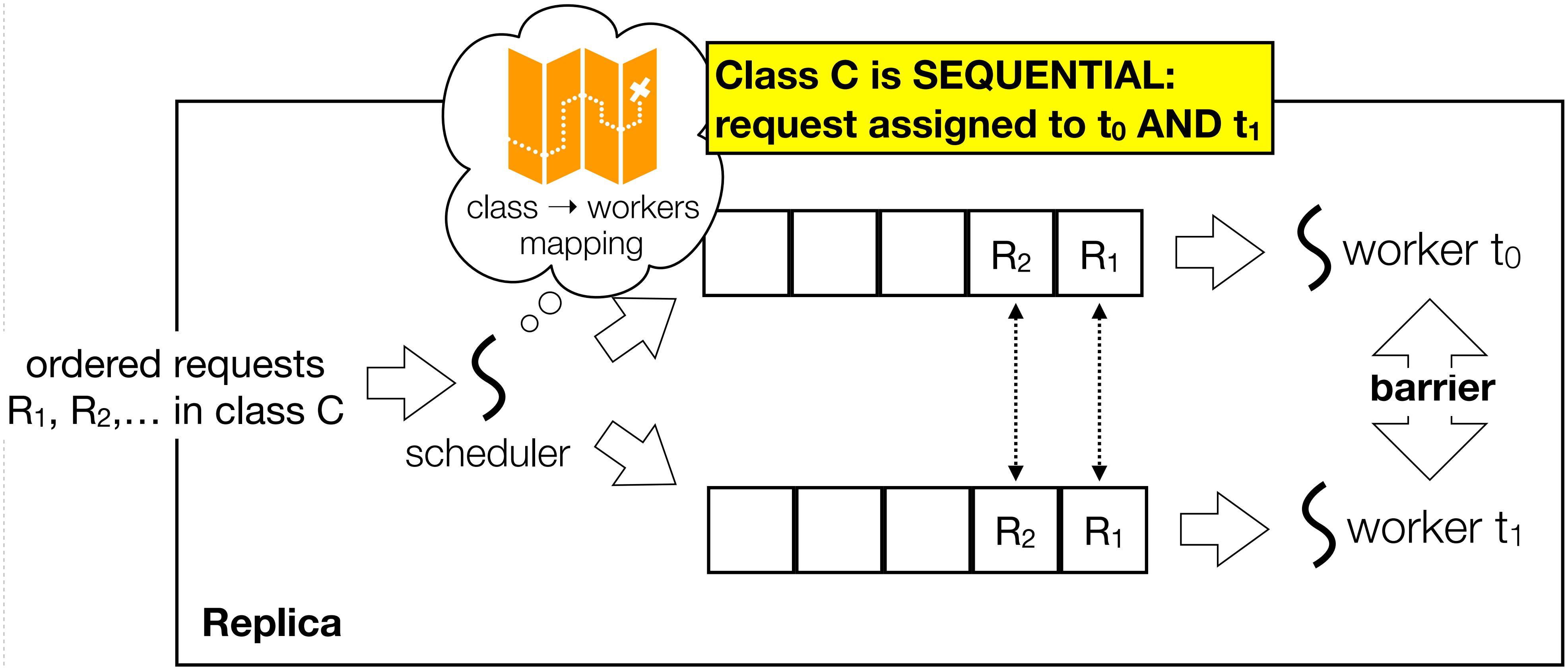
- Define workers that execute requests in the class
- Define class type
  - Sequential: one request at a time
  - Concurrent: requests executed concurrently



Early Scheduling execution model



Early Scheduling execution model



Mapping classes to workers

Rule #5

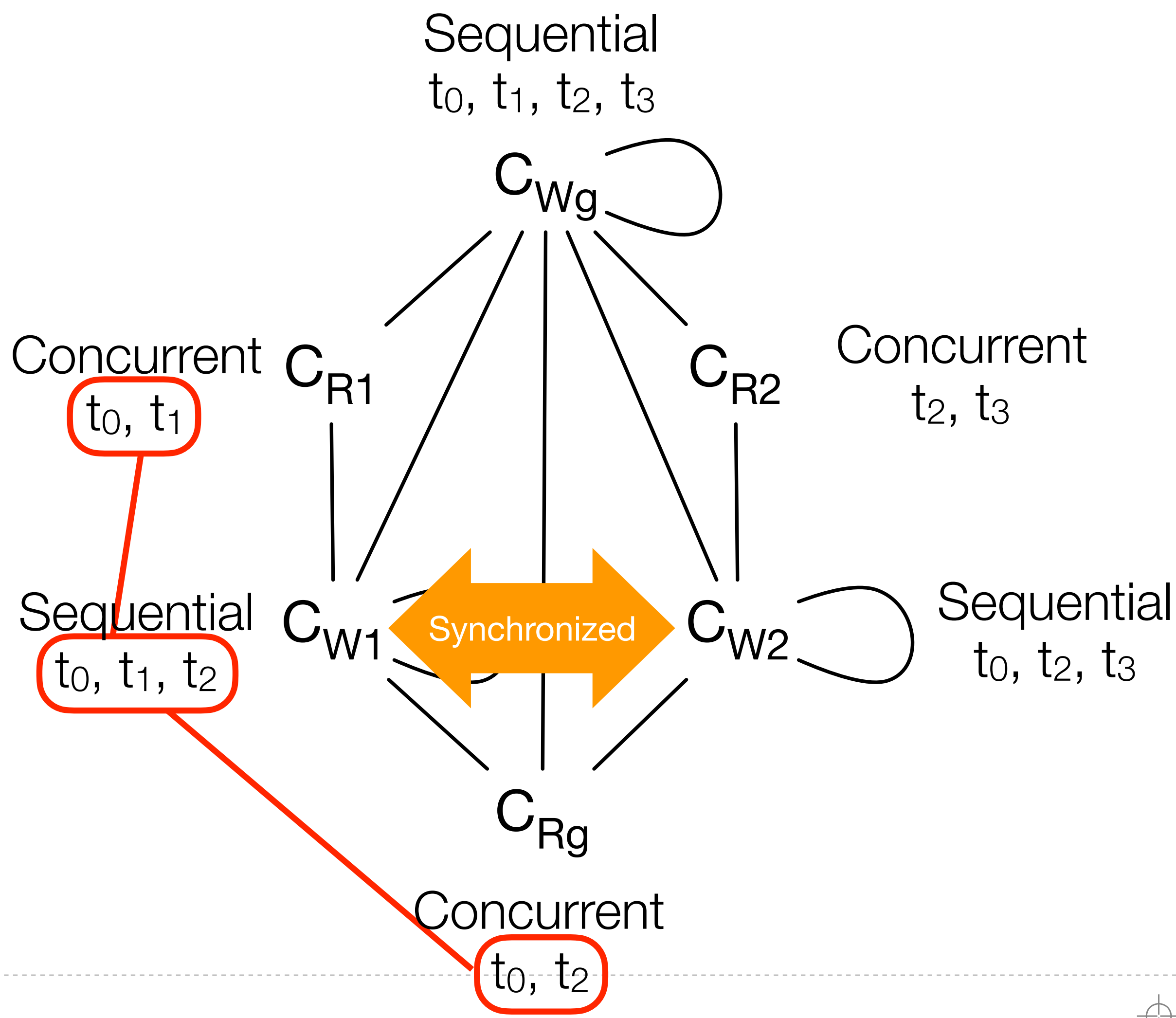
Sequential C_1 $\Rightarrow$ t_0,t_1,t_2

Sequential C_2 $\quad$ t_2,t_3,t_4

If C_1 and C_2 conflict, and are sequential, then C_1 and C_2 must have one worker in common

Mapping classes to workers

- Local reads most common requests
- Workers: t_0,t_1,t_2,t_3



Optimizing scheduling

- O_{1a}: Minimize workers in sequential classes
- O_{1b}: Maximize workers in concurrent classes
- O_2: Assign workers to concurrent classes in proportion to class weight (i.e., more work, more workers)
- O_3: Minimize unnecessary synchronization among classes

Optimization model

- Described in AMPL
- Solved with KNitro

Algorithm 3 Optimization model.

$\vdots$

constraints:

$\forall c \in C \ : \ \bigvee_{\forall t \in T} uses(c, t)$ // R.1

$\forall c \in C \ : \ \#[c_1, c_1] \Rightarrow Seq[c_1]$ // R.2

$\forall c_1, c_2 \in C \ : \ \#[c_1, c_2] \Rightarrow Seq[c_1] \vee Seq[c_2]$ // R.3

$\forall c_1, c_2 \in C, t \in T \ : \ \#[c_1, c_2] \wedge Seq[c_1] \wedge Cnc[c_2] \wedge uses[c_2, t] \Rightarrow uses[c_1, t]$ // R.4

$\forall c_1, c_2 \in C \ : \ \#[c_1, c_2] \wedge Seq[c_1] \wedge Seq[c_2] \Rightarrow \exists t \in T \ : \ uses[c_1, t] \wedge uses[c_2, t]$ // R.5

objective:

minimize cost:

$+ \sum_{\forall t \in T, \forall c \in C : Seq[c]} uses[c, t] \times w[c] \ / \ ws$ // O.1a

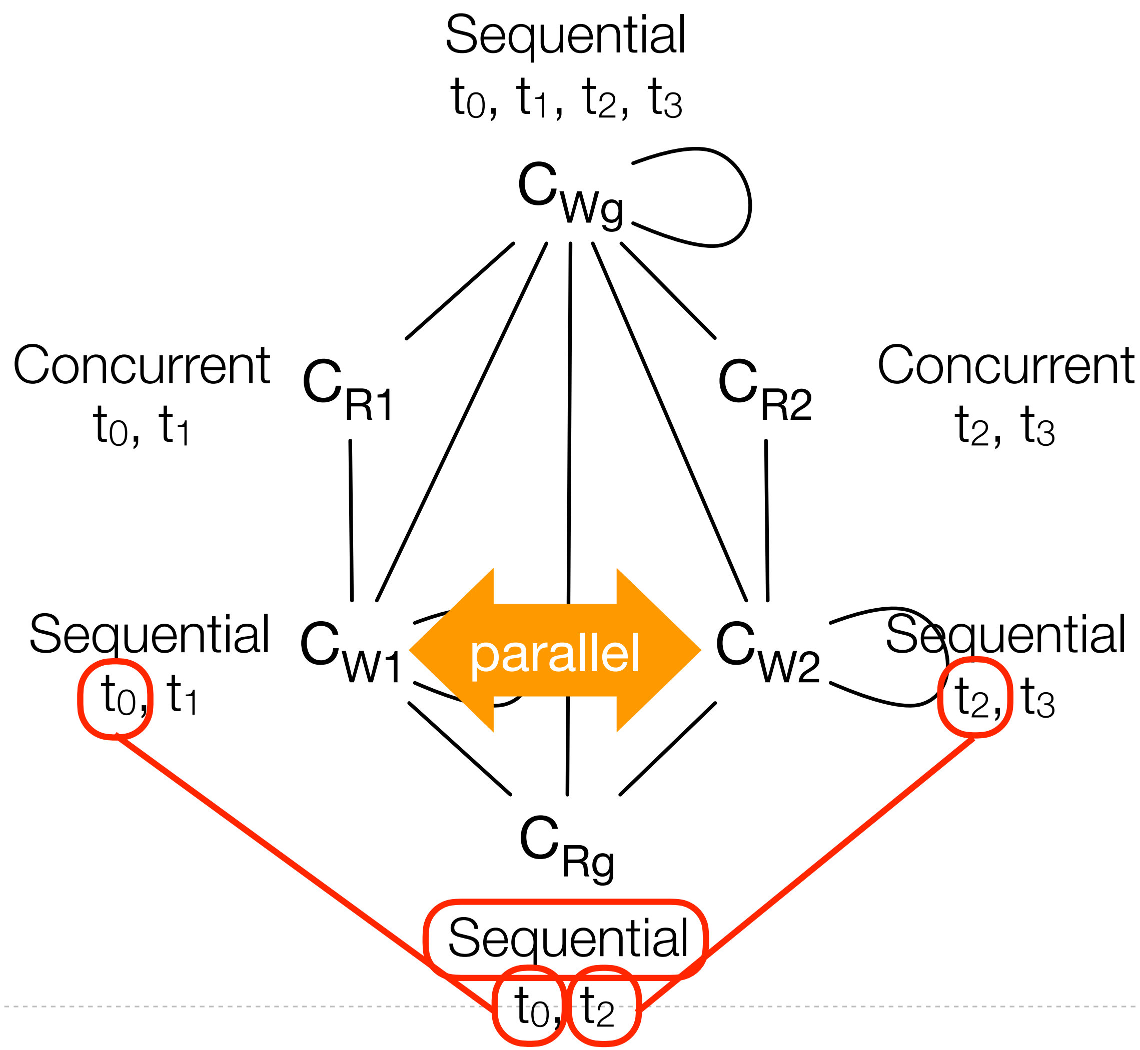
$- \sum_{\forall t \in T, \forall c \in C : Cnc[c]} uses[c, t] \times w[c] \ / \ wc$ // O.1b

$+ \sum_{\forall c \in C : Cnc[c]} |w[c] / wc - (|\{\forall t \in T : uses[c, t]\}| / nt)|$ // O.2

$+ \sum_{\forall c_1, c_2 \in C : Seq[c_1] \wedge Seq[c_2] \wedge \neg\#[c_1, c_2]} |\{\forall t \in T \ : \ uses[c_1, t] \wedge uses[c_2, t]\}| \times nt \times nc$ // O.3

Naive vs Optimized mapping

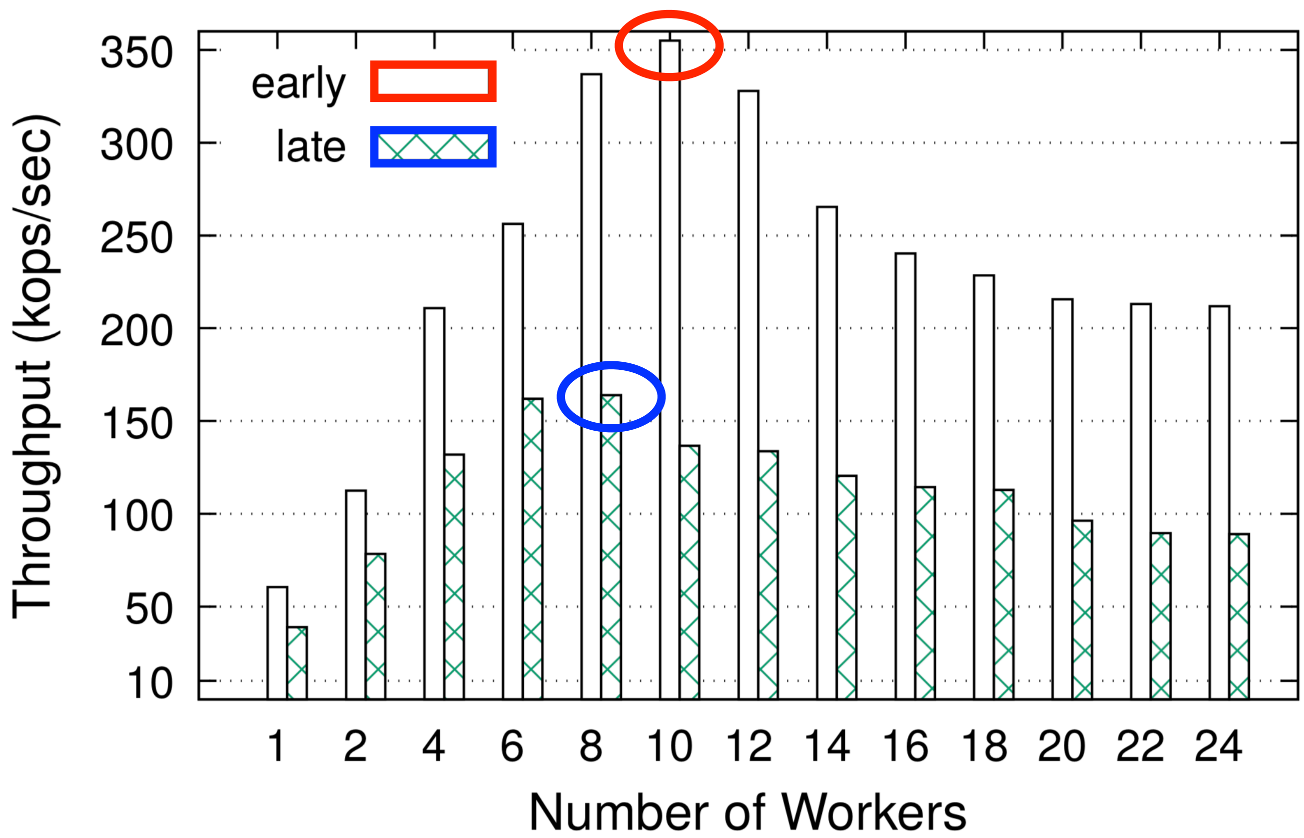
- **Local reads most common requests**
- **Workers: t_0,t_1,t_2,t_3**



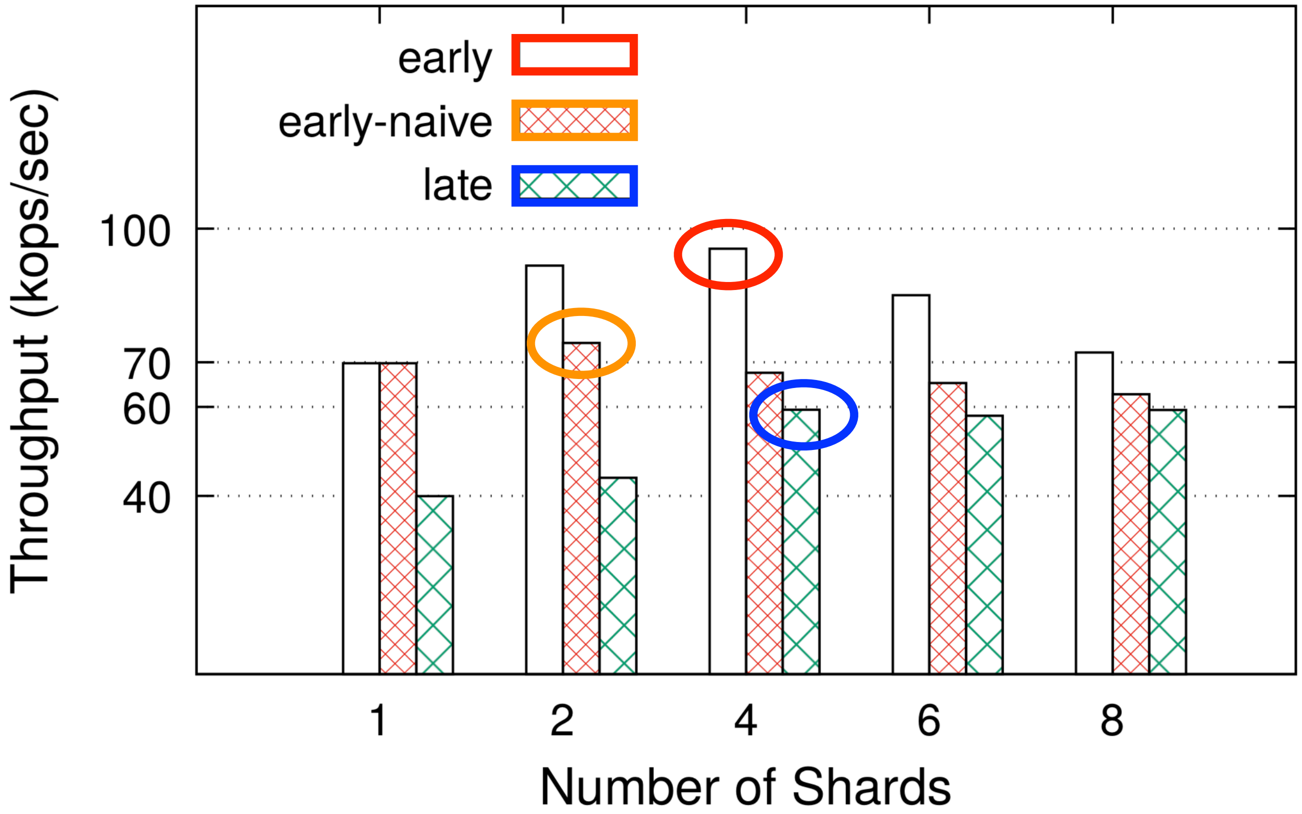
Experimental evaluation

- Prototype in BFT-SMaRt environment
  - Early scheduling and late scheduling
  - Configured to crash failures (not BFT)
- Linked-list application
  - Single- and multi-shard deployments
  - Light, moderate, and heavy execution costs
  - Uniform and skewed workloads

Single-shard, reads, moderate



Multi-shard, mixed, moderate



http://www.inf.usi.ch/faculty/pedone/