On the Implementation of an Or-Parallel Prolog System for Clusters of Multicores

João Santos and **Ricardo Rocha**

CRACS & INESC TEC and Faculty of Sciences
University of Porto, Portugal

ICLP'16 – October 20, 2016

Motivation

- The inherent non-determinism in the way Prolog programs are structured makes it very attractive for the exploitation of **implicit parallelism**. Prolog offers two major forms of implicit parallelism:
  - **And-Parallelism**
  - **Or-Parallelism**

Motivation

- The inherent non-determinism in the way Prolog programs are structured makes it very attractive for the exploitation of **implicit parallelism**. Prolog offers two major forms of implicit parallelism:
  - **And-Parallelism**
  - **Or-Parallelism**

- Many parallel Prolog systems/models have been developed in the past. However, none of them was specially designed to **explore both shared and distributed memory** in order to take advantage of the recent and popular architectures based on **clusters of multicores**.

Previous Work

- In previous work **[ComSIS 2014]**, we have proposed a **novel or-parallel model** to address the problem of efficiently exploit the combination of shared and distributed memory architectures.

Previous Work

- In previous work **[ComSIS 2014]**, we have proposed a **novel or-parallel model** to address the problem of efficiently exploit the combination of shared and distributed memory architectures.
- It introduces a **layered model with two scheduling levels**:
  - One for **workers sharing memory resources**, named a team of workers
  - Another for **teams of workers** (not sharing memory resources)

Previous Work

- In previous work **[ComSIS 2014]**, we have proposed a **novel or-parallel model** to address the problem of efficiently exploit the combination of shared and distributed memory architectures.
- It introduces a **layered model with two scheduling levels**:
  - One for **workers sharing memory resources**, named a team of workers
  - Another for **teams of workers** (not sharing memory resources)
- But, it only describes the high-level algorithms that support the key aspects of the layered model (**not any implementation**).

Main Contributions

- In this work, we describe a first **implementation of the layered model** and focus the description on the operational aspects of our specific implementation of the model:
  - How a parallel engine is created
  - How a parallel goal is launched
  - How the team scheduler is structured in different modules
  - How we have implemented the sharing work process between teams
  - How we deal with load balancing and termination
  - ...

Main Contributions

- In this work, we describe a first **implementation of the layered model** and focus the description on the operational aspects of our specific implementation of the model:
  - How a parallel engine is created
  - How a parallel goal is launched
  - How the team scheduler is structured in different modules
  - How we have implemented the sharing work process between teams
  - How we deal with load balancing and termination
  - ...
- We propose a **new set of built-in predicates** that constitute the syntax to interact with an or-parallel engine in our implementation.

Main Contributions

- In this work, we describe a first **implementation of the layered model** and focus the description on the operational aspects of our specific implementation of the model:
  - How a parallel engine is created
  - How a parallel goal is launched
  - How the team scheduler is structured in different modules
  - How we have implemented the sharing work process between teams
  - How we deal with load balancing and termination
  - ...
- We propose a **new set of built-in predicates** that constitute the syntax to interact with an or-parallel engine in our implementation.
- We show experimental results with different configurations of **teams up to 32 workers**.

Background: Environment Copying

- When implementing or-parallelism, a main difficulty is how to efficiently represent the **multiple bindings** for the same variable produced by the parallel execution of alternative matching clauses.

Background: Environment Copying

- When implementing or-parallelism, a main difficulty is how to efficiently represent the **multiple bindings** for the same variable produced by the parallel execution of alternative matching clauses.
- One of the most successful or-parallel models that solves the multiple bindings problem is **environment copying**, which has been efficiently used in the implementation of or-parallel systems **both on shared memory and on distributed memory architectures**.

Background: Environment Copying

- When implementing or-parallelism, a main difficulty is how to efficiently represent the **multiple bindings** for the same variable produced by the parallel execution of alternative matching clauses.
- One of the most successful or-parallel models that solves the multiple bindings problem is **environment copying**, which has been efficiently used in the implementation of or-parallel systems **both on shared memory and on distributed memory architectures**.
- With environment copying model, **each worker keeps a separate copy of its own environment**, thus the bindings to shared variables are done as usual and without conflicts. Advantages:
  - Each worker acts like an independent Prolog engine
  - Requires minimal synchronization between workers

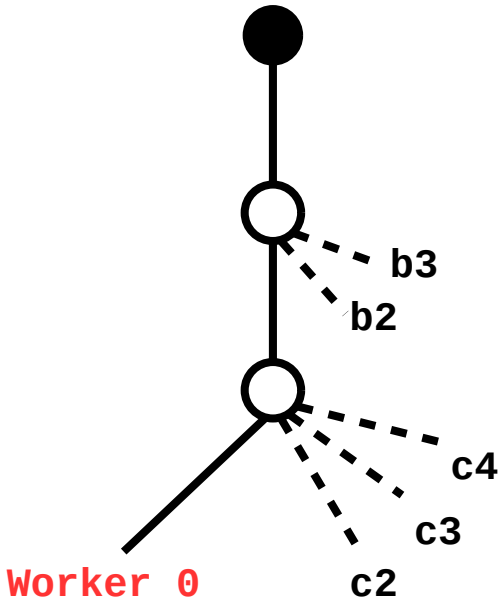
Background: Environment Copying

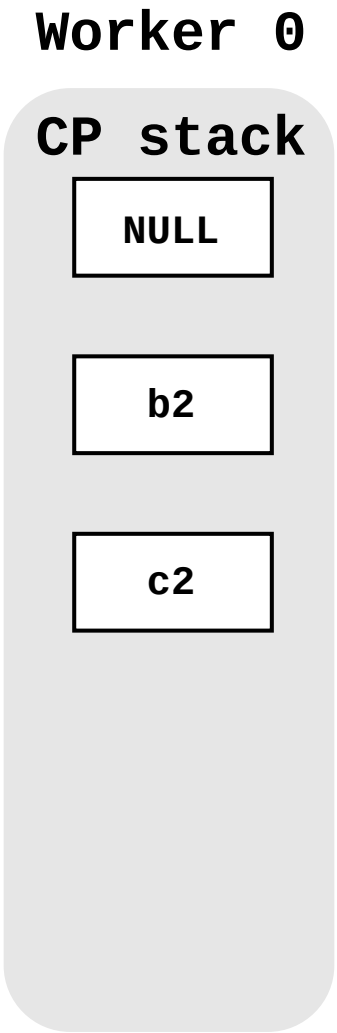
- Arguably, the most successful systems/models using environment copying are:
  - **Muse** [Ali and Karlsson, 1990] for shared memory architectures
  - **PALS** [Villaverde, Pontelli, Guo and Gupta, 2001] for distributed memory architectures

Background: Environment Copying

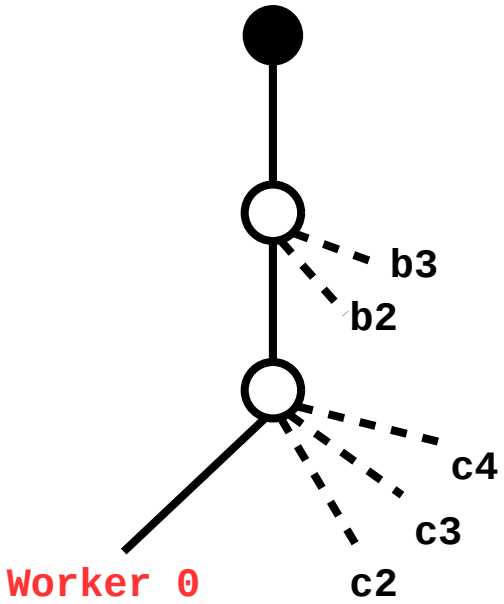
- Arguably, the most successful systems/models using environment copying are:
  - **Muse** [Ali and Karlsson, 1990] for shared memory architectures
  - **PALS** [Villaverde, Pontelli, Guo and Gupta, 2001] for distributed memory architectures
- The main difference between both is on how **scheduling** is done:
  - Muse uses **dynamic distribution of work** (**or-frames**)
  - PALS uses **static distribution of work** (**stack splitting**)

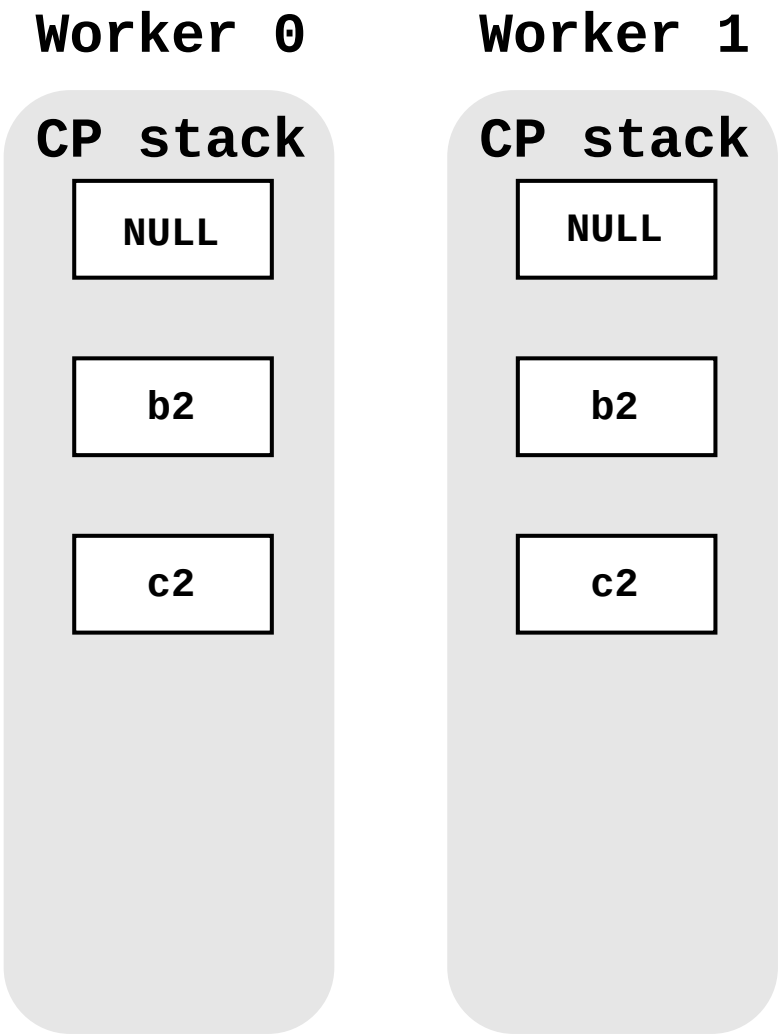
Background: Or-Frames



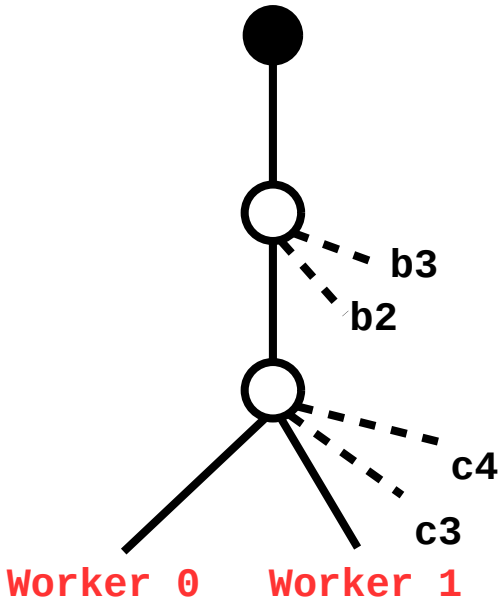


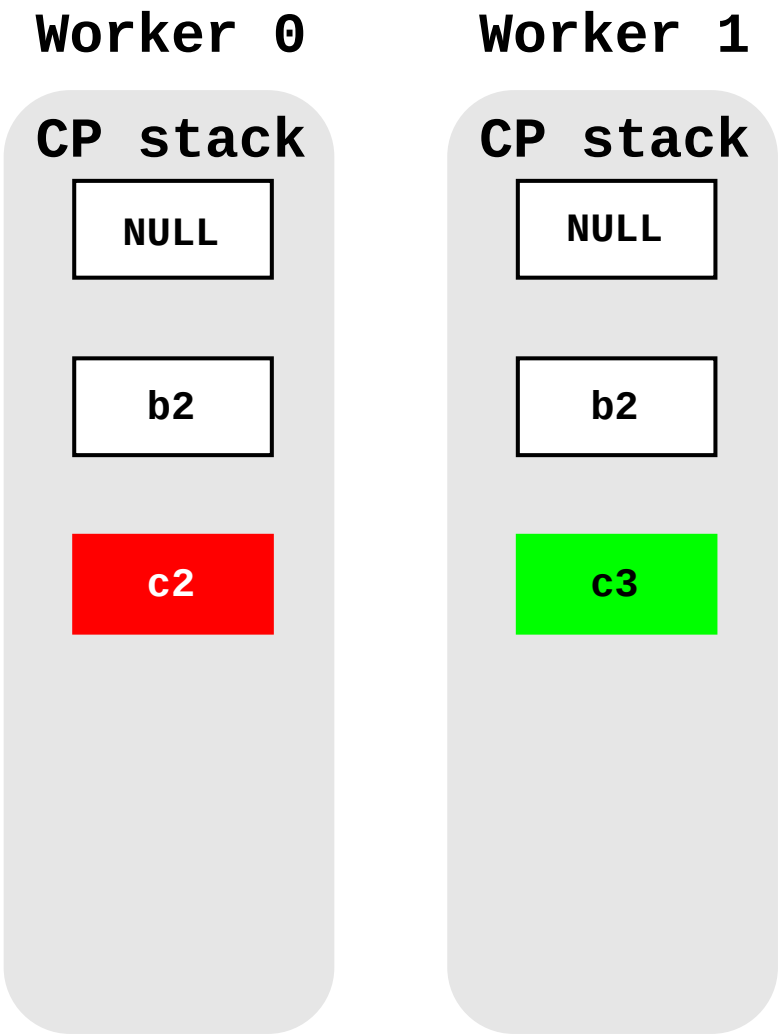
Background: Or-Frames



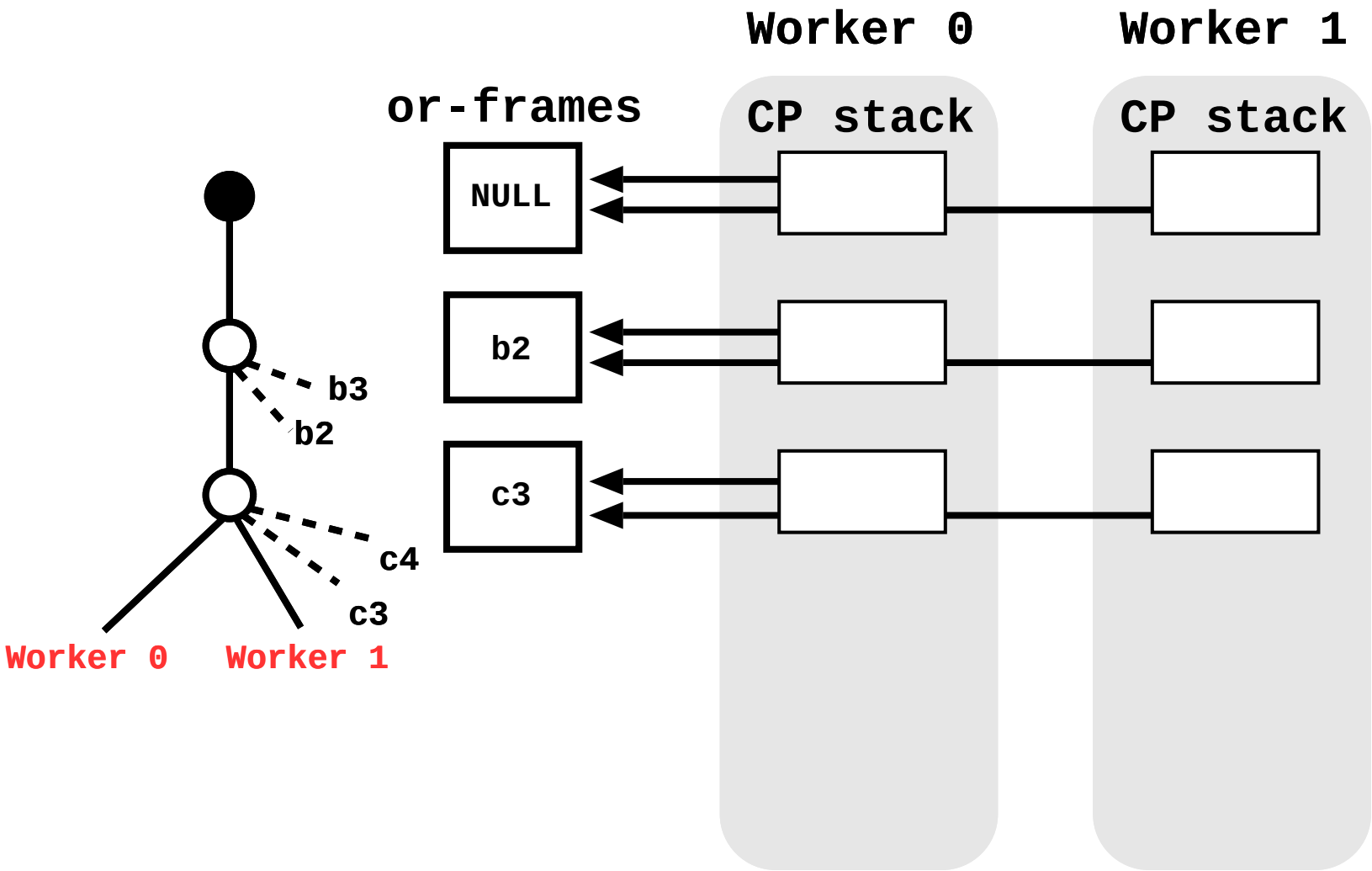


Background: Or-Frames

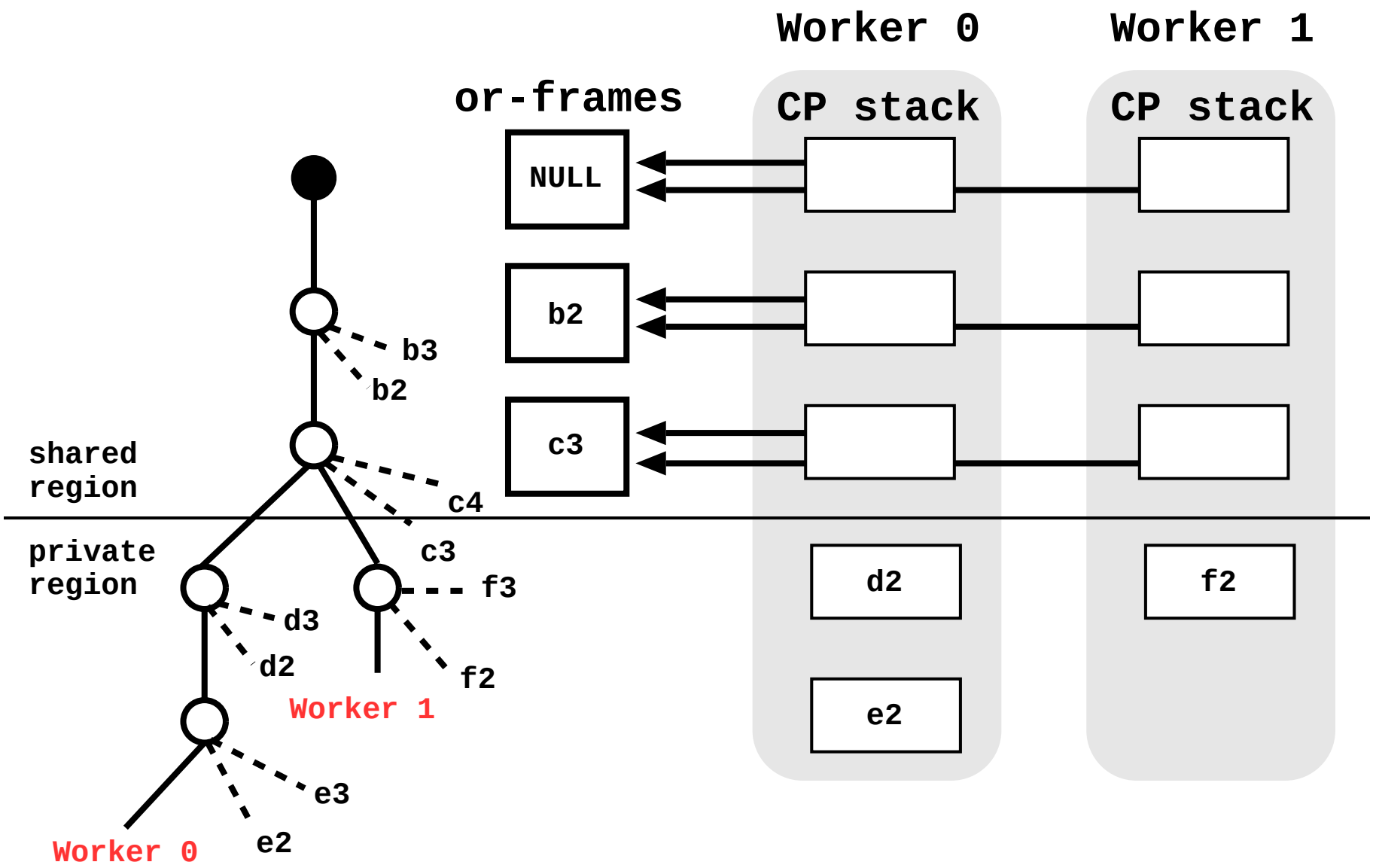




Background: Or-Frames

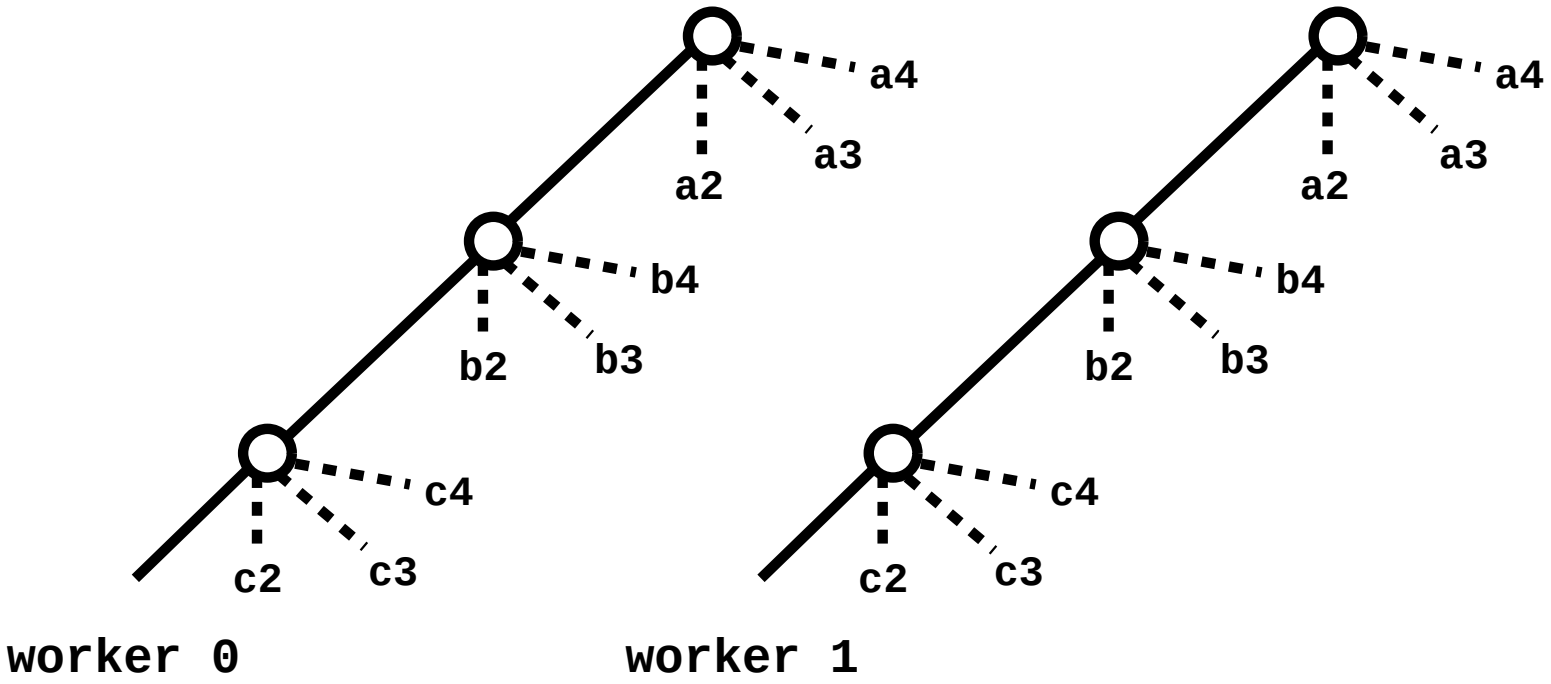


Background: Or-Frames



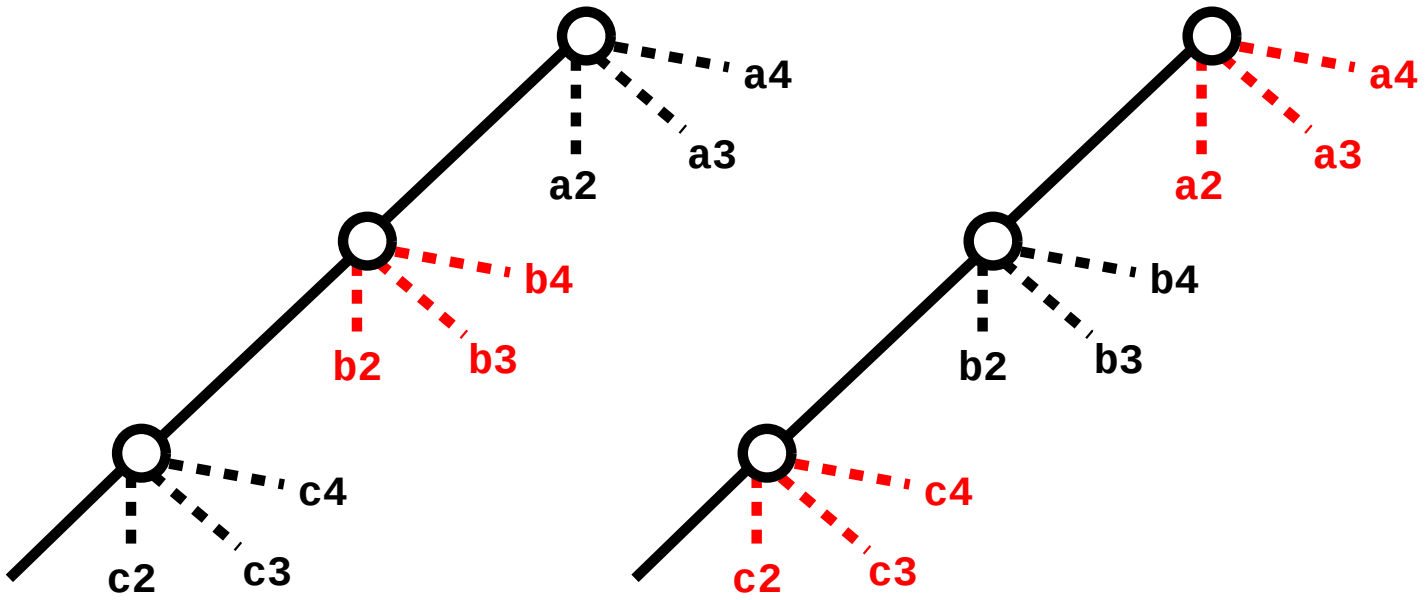
Background: Stack Splitting

- After environment copying



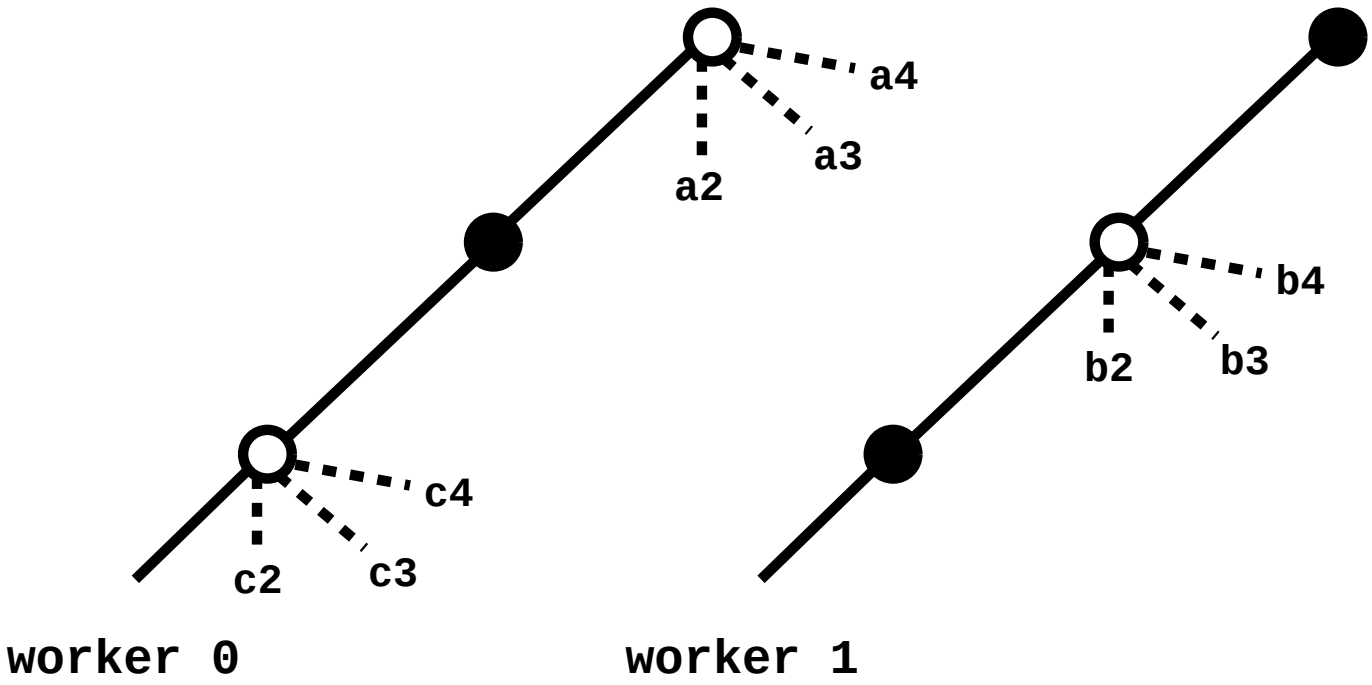
Background: Stack Splitting

- Vertical splitting



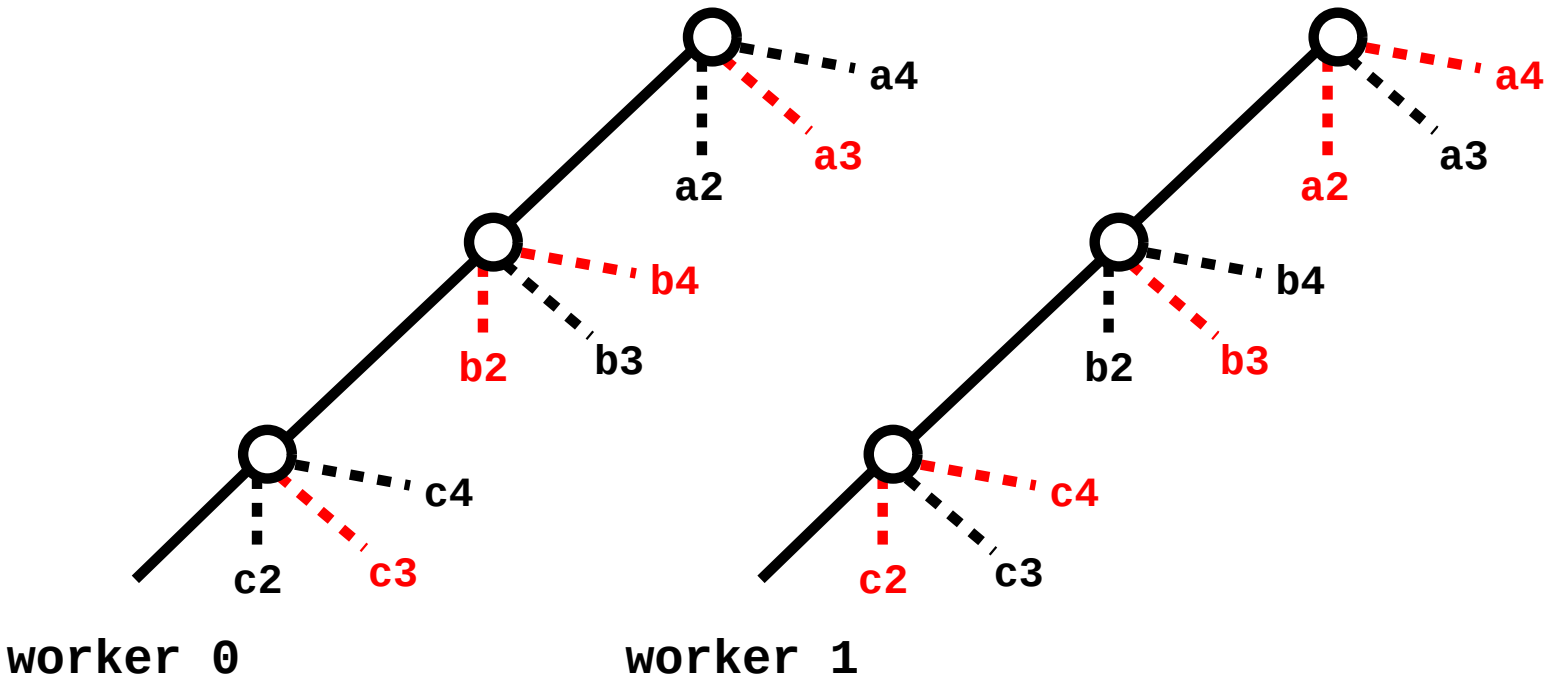
Background: Stack Splitting

- Vertical splitting



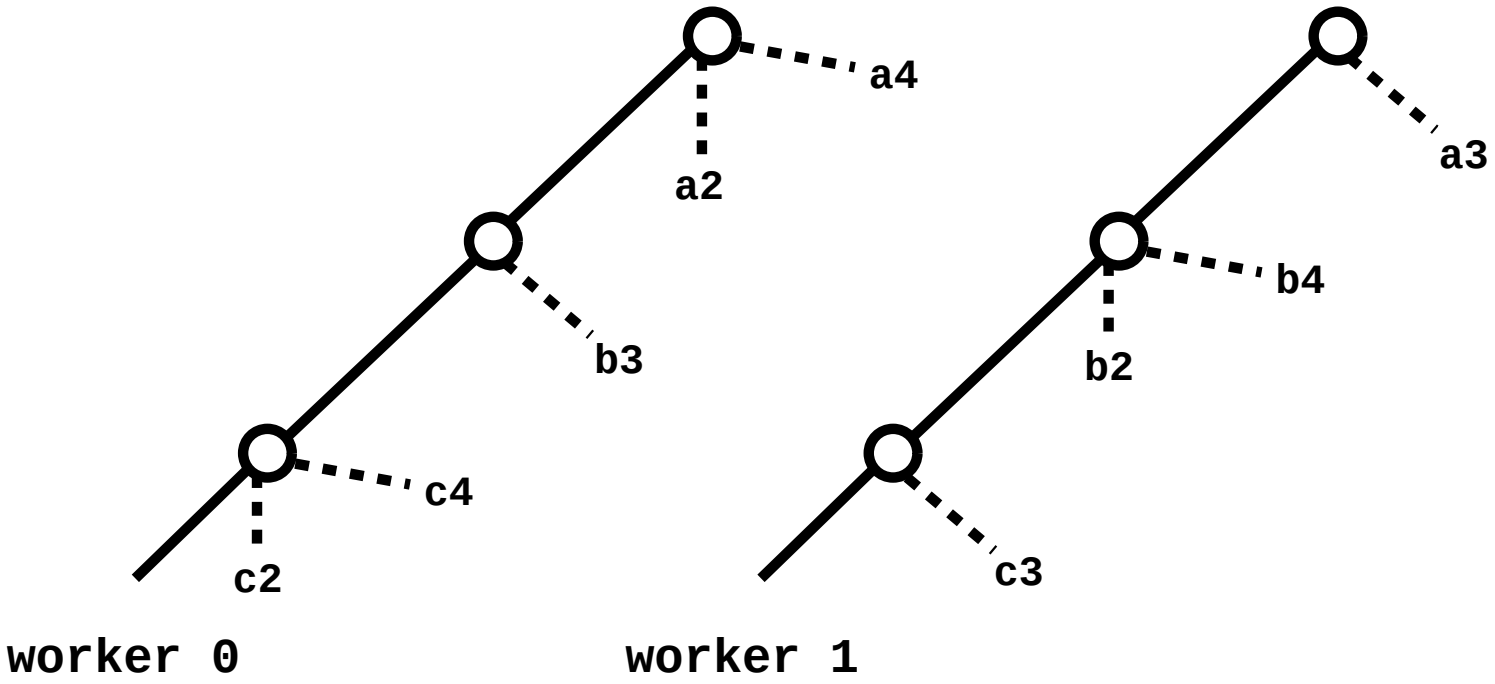
Background: Stack Splitting

- Horizontal splitting



Background: Stack Splitting

- Horizontal splitting



Layered Model

- Introduces the concept of **teams** and the ability to **exploit different scheduling strategies** for distributing work among teams and among the workers inside a team (**two-level scheduler**).

Layered Model

- Introduces the concept of **teams** and the ability to **exploit different scheduling strategies** for distributing work among teams and among the workers inside a team (**two-level scheduler**).
  - Defines a team as a set of workers (processes or threads) who **share the same memory address space** and cooperate to solve a certain part of the main problem.

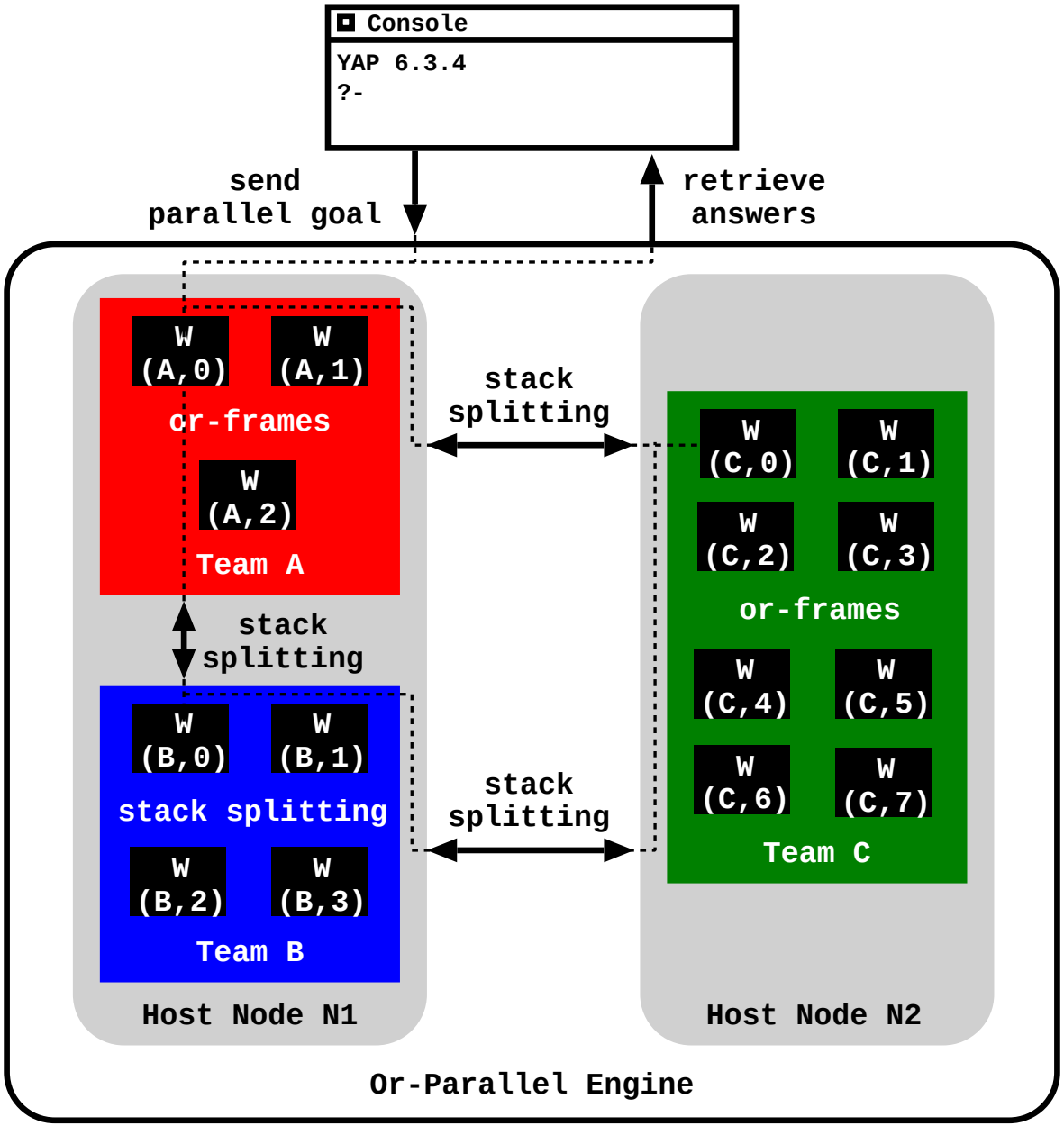
Layered Model

- Introduces the concept of **teams** and the ability to **exploit different scheduling strategies** for distributing work among teams and among the workers inside a team (**two-level scheduler**).
  - Defines a team as a set of workers (processes or threads) who **share the same memory address space** and cooperate to solve a certain part of the main problem.
  - **Inside a team** we may use or-frames or stack splitting to distribute work.

Layered Model

- Introduces the concept of **teams** and the ability to **exploit different scheduling strategies** for distributing work among teams and among the workers inside a team (**two-level scheduler**).
  - Defines a team as a set of workers (processes or threads) who **share the same memory address space** and cooperate to solve a certain part of the main problem.
  - **Inside a team** we may use or-frames or stack splitting to distribute work.
  - **Between teams** we use a new layer to distribute work using stack splitting between teams, but we can still use or-frames for distributing work among the workers inside a team.

Layered Model



New Syntax

- Our proposal for the syntax to interact with an or-parallel engine follows two important design rules:
  - **Avoid blocking mechanisms** for interacting with an or-parallel engine
  - **Delegate** to the programmer the responsibility of **explicitly annotate** which parts of the program should be run in an or-parallel engine

New Syntax

- Our proposal for the syntax to interact with an or-parallel engine follows two important design rules:
  - **Avoid blocking mechanisms** for interacting with an or-parallel engine
  - **Delegate** to the programmer the responsibility of **explicitly annotate** which parts of the program should be run in an or-parallel engine
- Predicates:
  - `par_create_parallel_engine(EngName, ListTeams)`
  - `par_run_goal(EngName, Goal, Template)`
  - `par_probe_answers(EngName)`
  - `par_get_answers(EngName, Mode, ListAnswers, NumAnswers)`
  - `par_free_parallel_engine(EngName)`

Our Implementation

- To implement the layered model, we **extended the YapOr system** to efficiently exploit parallelism between teams of workers running on top of clusters of multicores.
  - YapOr is an or-parallel engine based on the environment copying (or-frames) model that extends the Yap Prolog system to exploit implicit or-parallelism in shared memory architectures
  - Our implementation takes full advantage of Yap's state-of-the-art engine and reuses the underlying execution environment, scheduler and part of the data structures used to support parallelism in YapOr.

Our Implementation

- To implement the layered model, we **extended the YapOr system** to efficiently exploit parallelism between teams of workers running on top of clusters of multicores.
  - YapOr is an or-parallel engine based on the environment copying (or-frames) model that extends the Yap Prolog system to exploit implicit or-parallelism in shared memory architectures
  - Our implementation takes full advantage of Yap's state-of-the-art engine and reuses the underlying execution environment, scheduler and part of the data structures used to support parallelism in YapOr.
- Each team is implemented as an independent YapOr engine using **or-frames**. For sharing work among teams, we implemented a **second-layer** that may use **vertical or horizontal splitting**.

Execution Model

- In order to allow the parallel execution of goals, it is necessary to create beforehand, at least, one or-parallel engine (we can create several or-parallel engines and run different parallel goals on each).
  - The worker 0 of each team is named the **master worker** of the team and it is responsible for launching the execution inside the team and for the communication with the other teams.
  - The first team to be launched is named the **master team** and its master worker is also responsible for launching the execution of the parallel goals and for returning the found answers.

Execution Model

- When the predicate **par_run_goal/3** is called in the client worker (Yap's console) then a message with the goal to be run in parallel is sent to the master worker of the master team.
  - It then notifies all the other master workers about the new parallel goal and then start its execution.
  - The other teams are now aware that a parallel computation has begun and thus they enter in **team scheduling mode**.

Execution Model

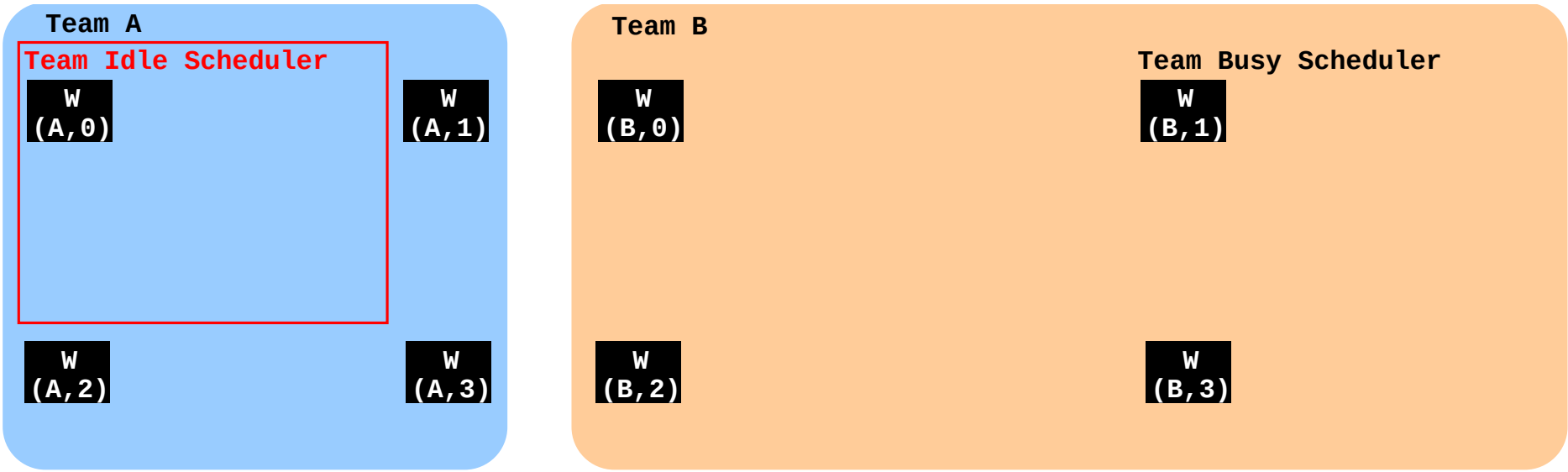
- When the predicate **par_run_goal/3** is called in the client worker (Yap's console) then a message with the goal to be run in parallel is sent to the master worker of the master team.
  - It then notifies all the other master workers about the new parallel goal and then start its execution.
  - The other teams are now aware that a parallel computation has begun and thus they enter in **team scheduling mode**.
- When a team has work, the execution behaves like an independent YapOr engine.
- A team is considered to be out of work when every worker inside the team is idle. The execution ends when all teams are idle.

Team Scheduler

- The team scheduler is divided in two modules:
  - The **team idle scheduler** is run by the master worker when a team is out of work and its goal is to find a busy team willing to share work.
  - The **team busy scheduler** is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams.

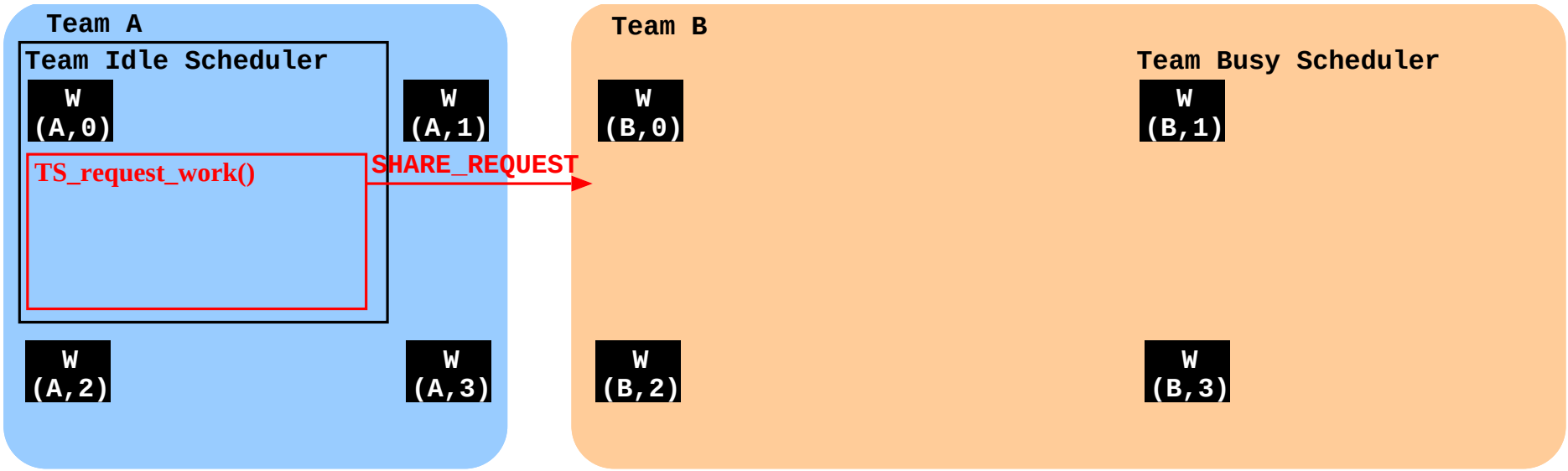
Team Scheduler

- The team scheduler is divided in two modules:
  - The **team idle scheduler** is run by the master worker when a team is out of work and its goal is to find a busy team willing to share work.
  - The **team busy scheduler** is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams.



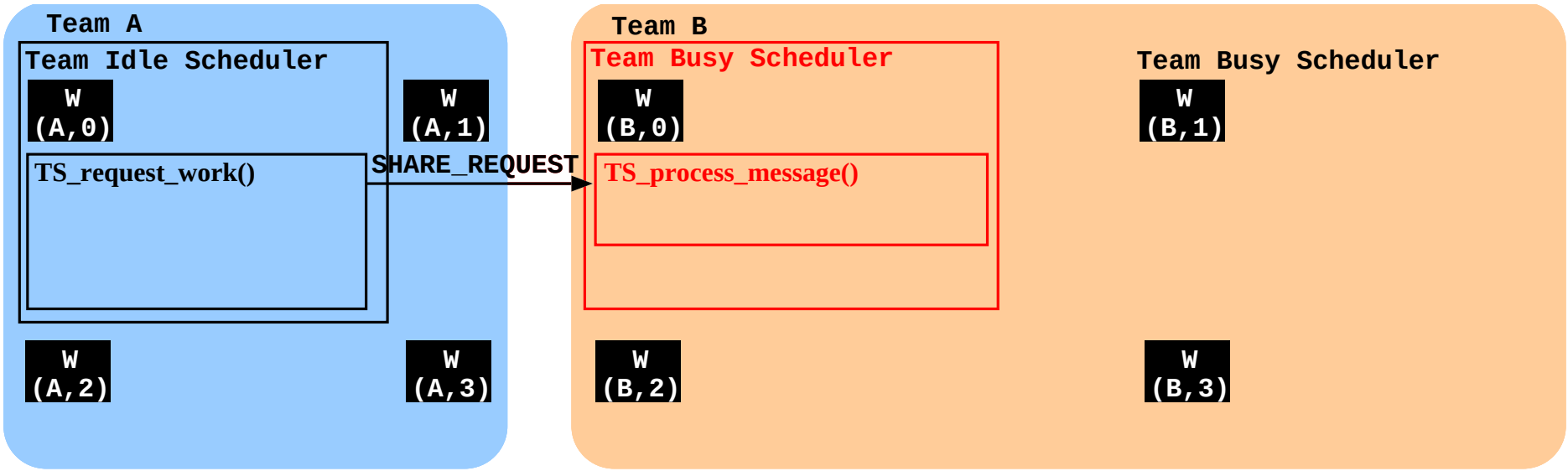
Team Scheduler

- The team scheduler is divided in two modules:
  - The **team idle scheduler** is run by the master worker when a team is out of work and its goal is to find a busy team willing to share work.
  - The **team busy scheduler** is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams.



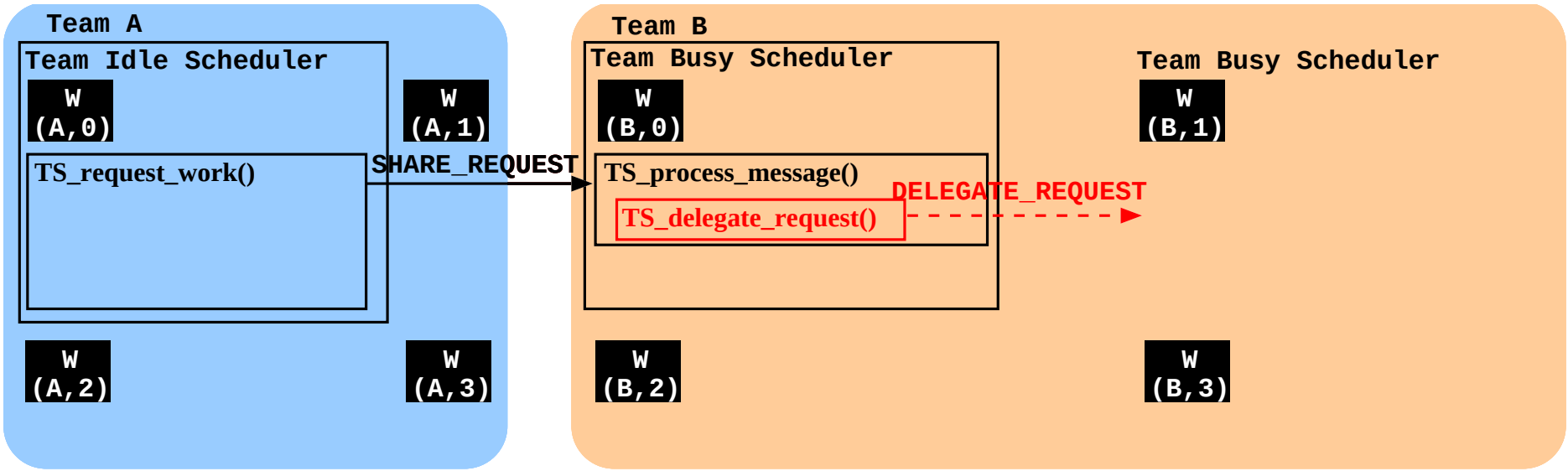
Team Scheduler

- The team scheduler is divided in two modules:
  - The **team idle scheduler** is run by the master worker when a team is out of work and its goal is to find a busy team willing to share work.
  - The **team busy scheduler** is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams.



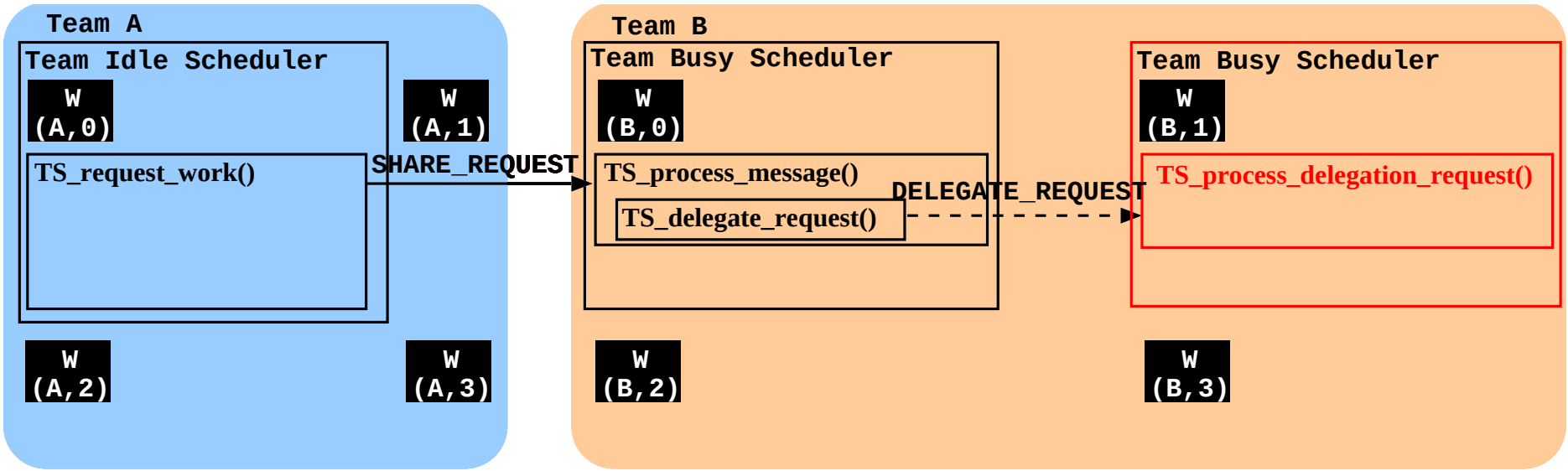
Team Scheduler

- The team scheduler is divided in two modules:
  - The **team idle scheduler** is run by the master worker when a team is out of work and its goal is to find a busy team willing to share work.
  - The **team busy scheduler** is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams.



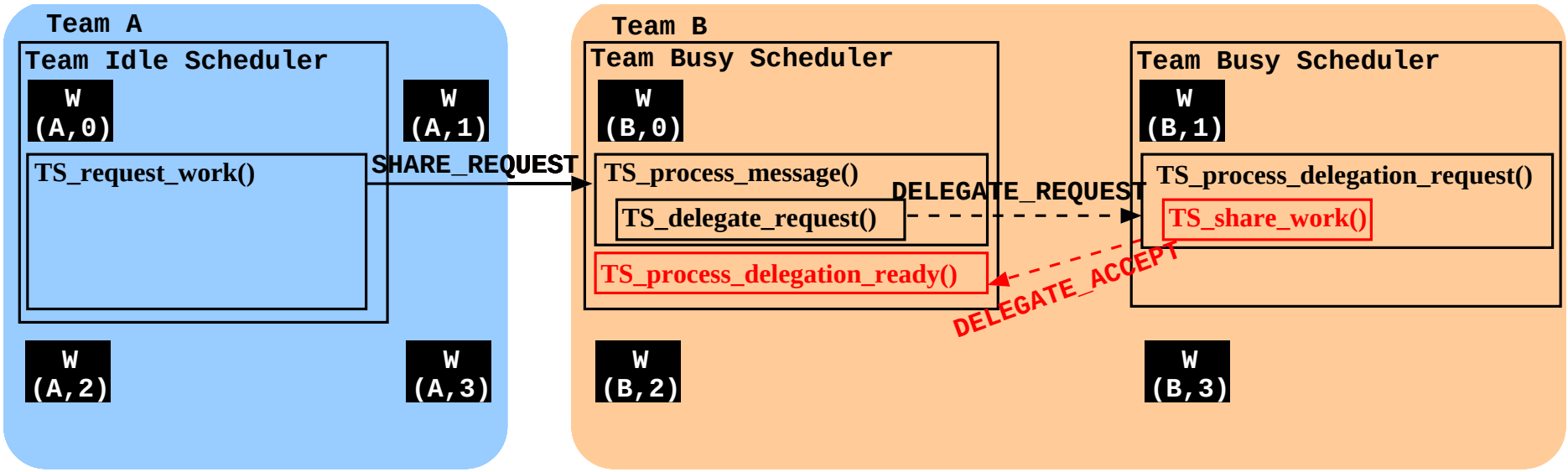
Team Scheduler

- The team scheduler is divided in two modules:
  - The **team idle scheduler** is run by the master worker when a team is out of work and its goal is to find a busy team willing to share work.
  - The **team busy scheduler** is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams.



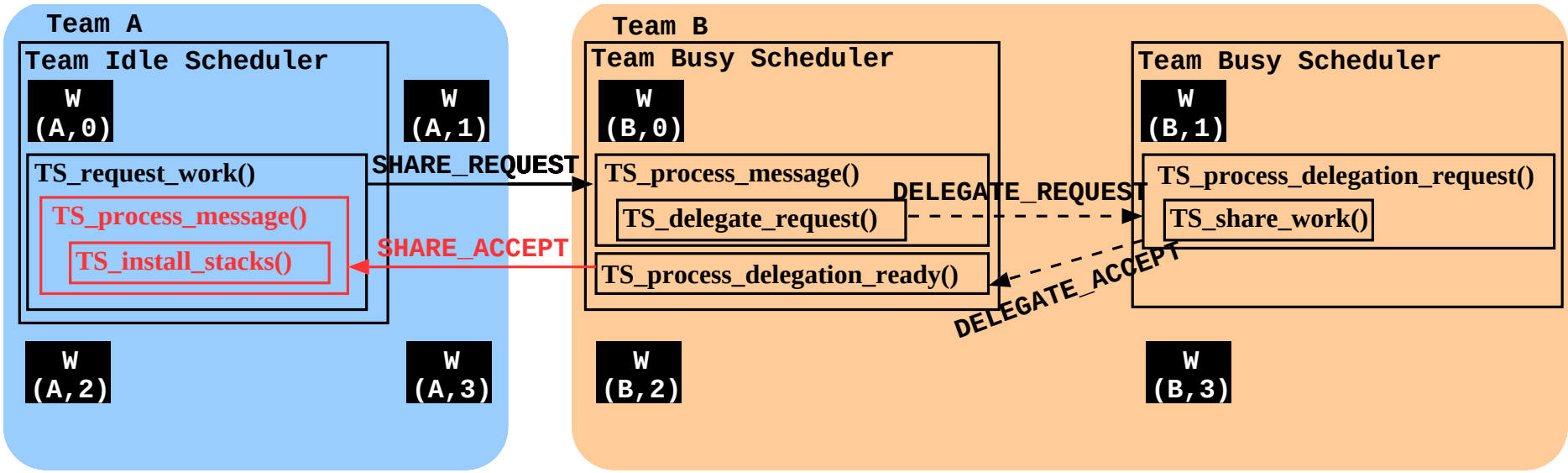
Team Scheduler

- The team scheduler is divided in two modules:
  - The **team idle scheduler** is run by the master worker when a team is out of work and its goal is to find a busy team willing to share work.
  - The **team busy scheduler** is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams.

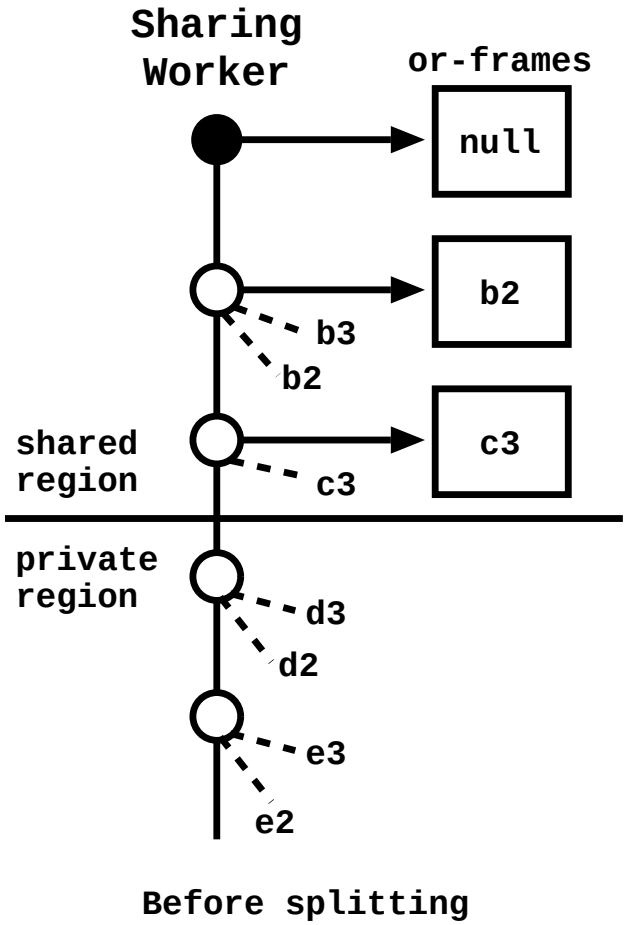


Team Scheduler

- The team scheduler is divided in two modules:
  - The **team idle scheduler** is run by the master worker when a team is out of work and its goal is to find a busy team willing to share work.
  - The **team busy scheduler** is run from time to time by all workers inside a busy team and it is responsible for handling the sharing requests sent by the idle teams.

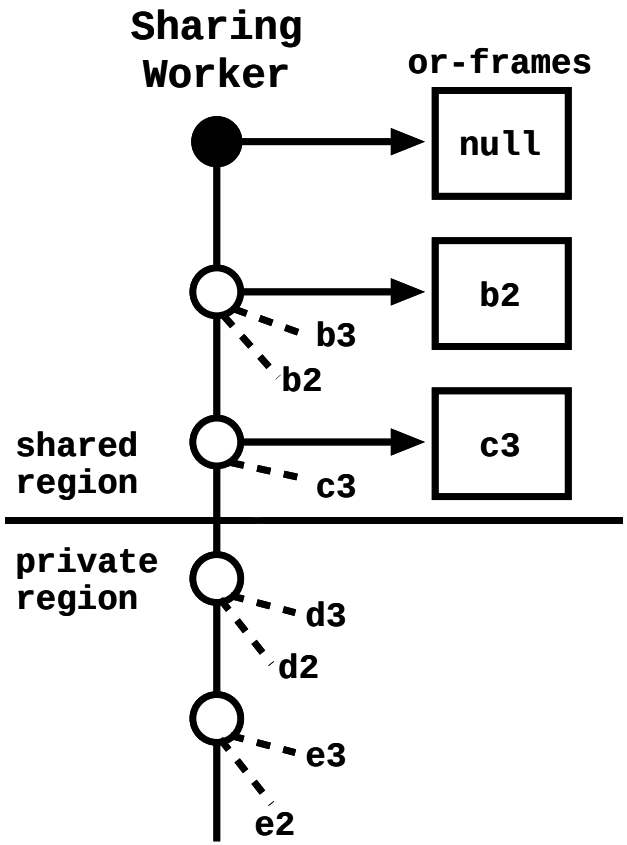


Vertical Stack Splitting Between Teams

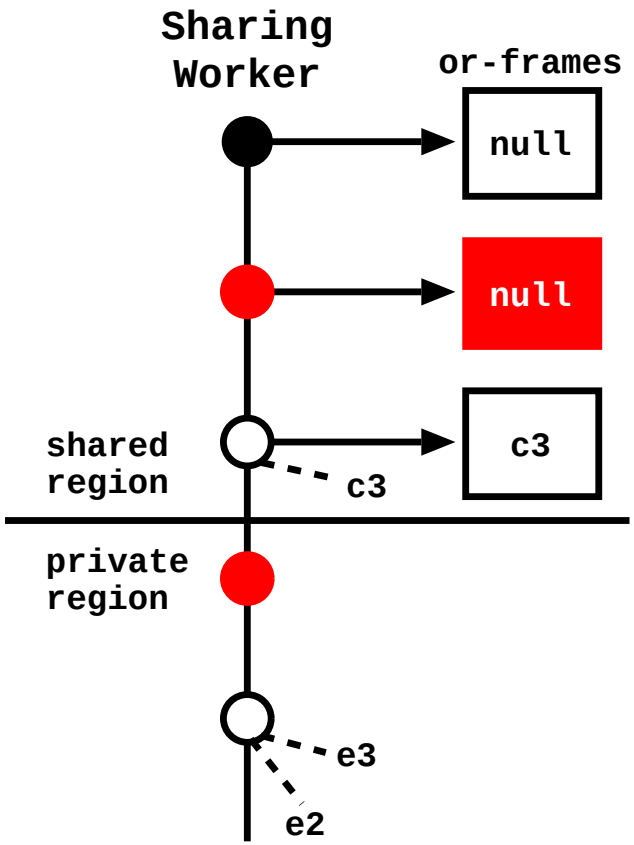


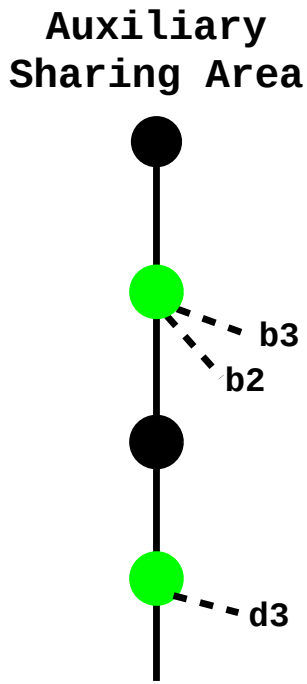
Before splitting

Vertical Stack Splitting Between Teams



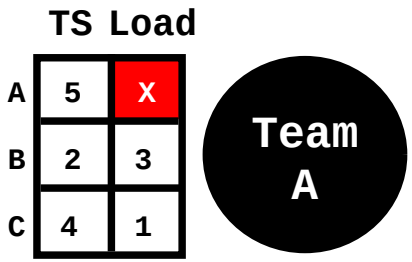
Before splitting

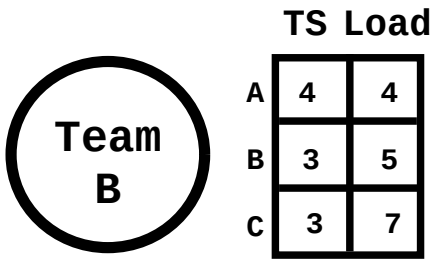




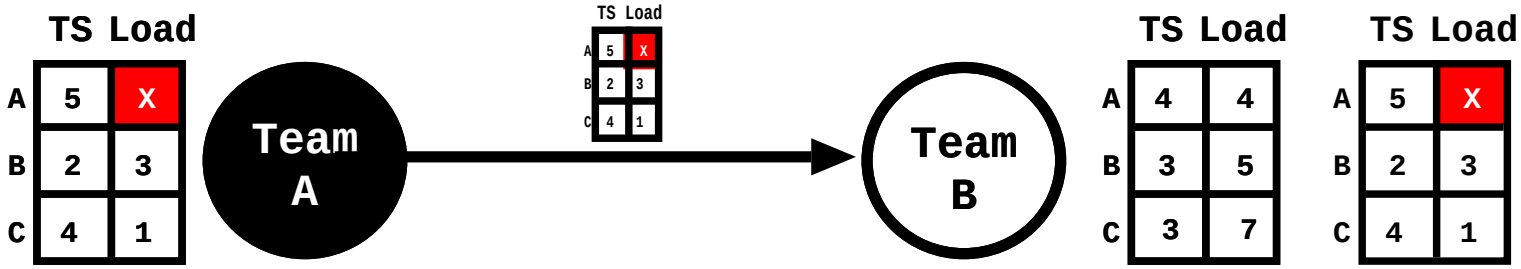
After splitting

Load Balancing With Timestamps

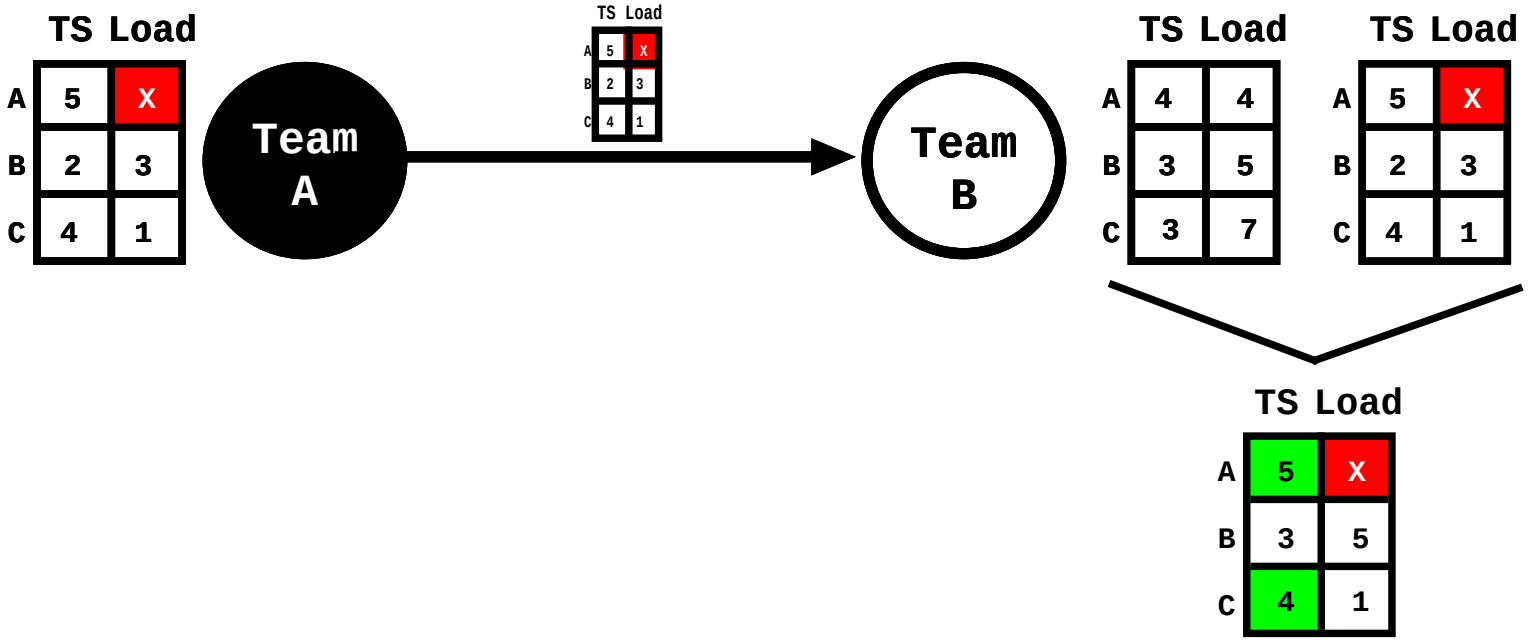




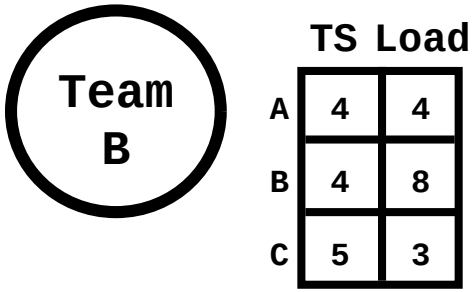
Load Balancing With Timestamps

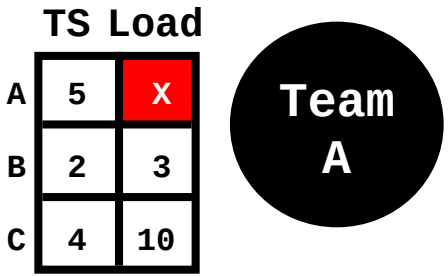


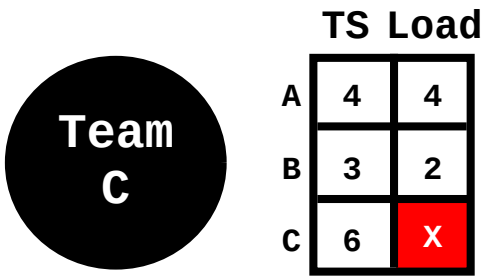
Load Balancing With Timestamps



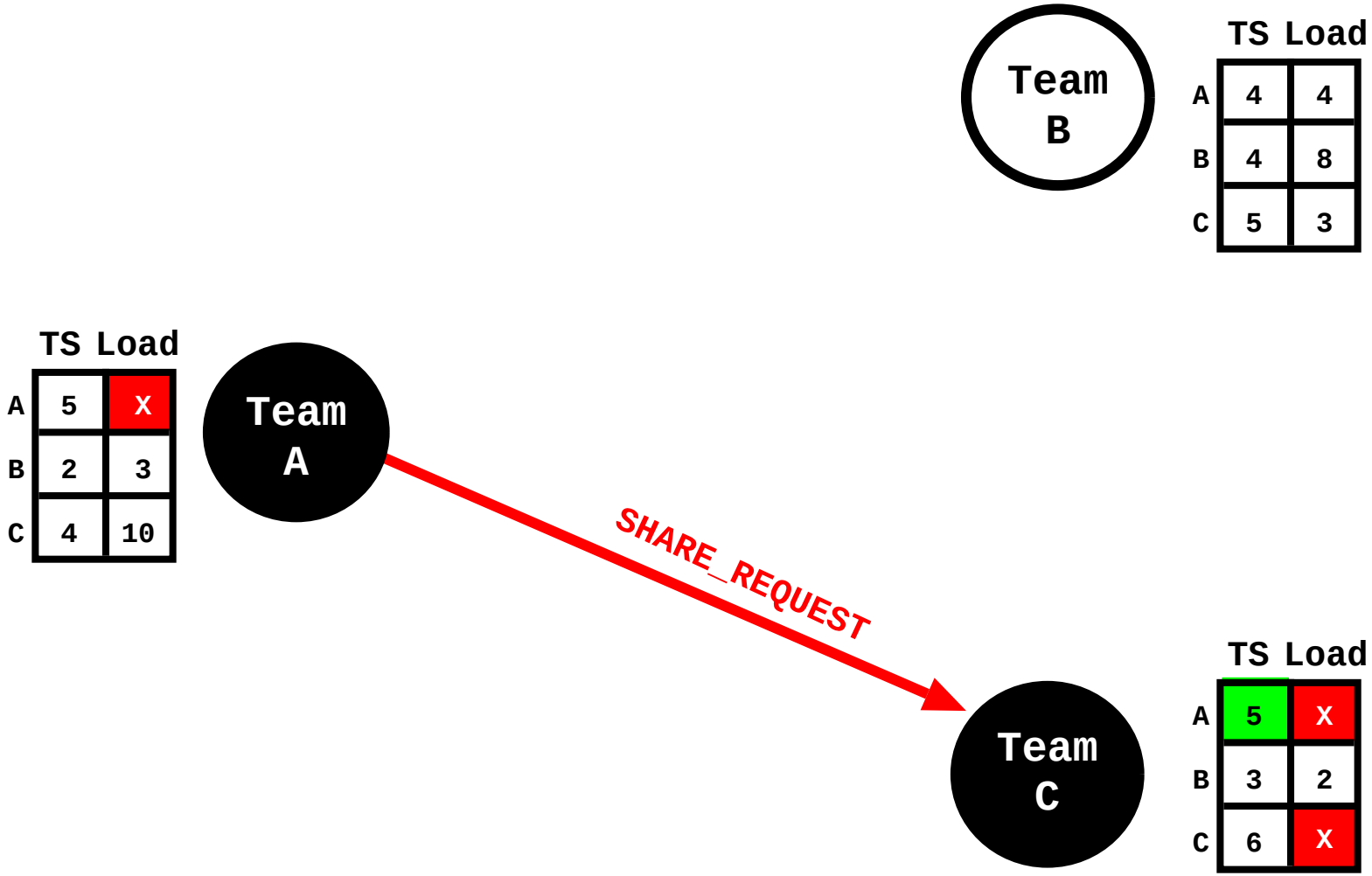
Termination With Timestamps



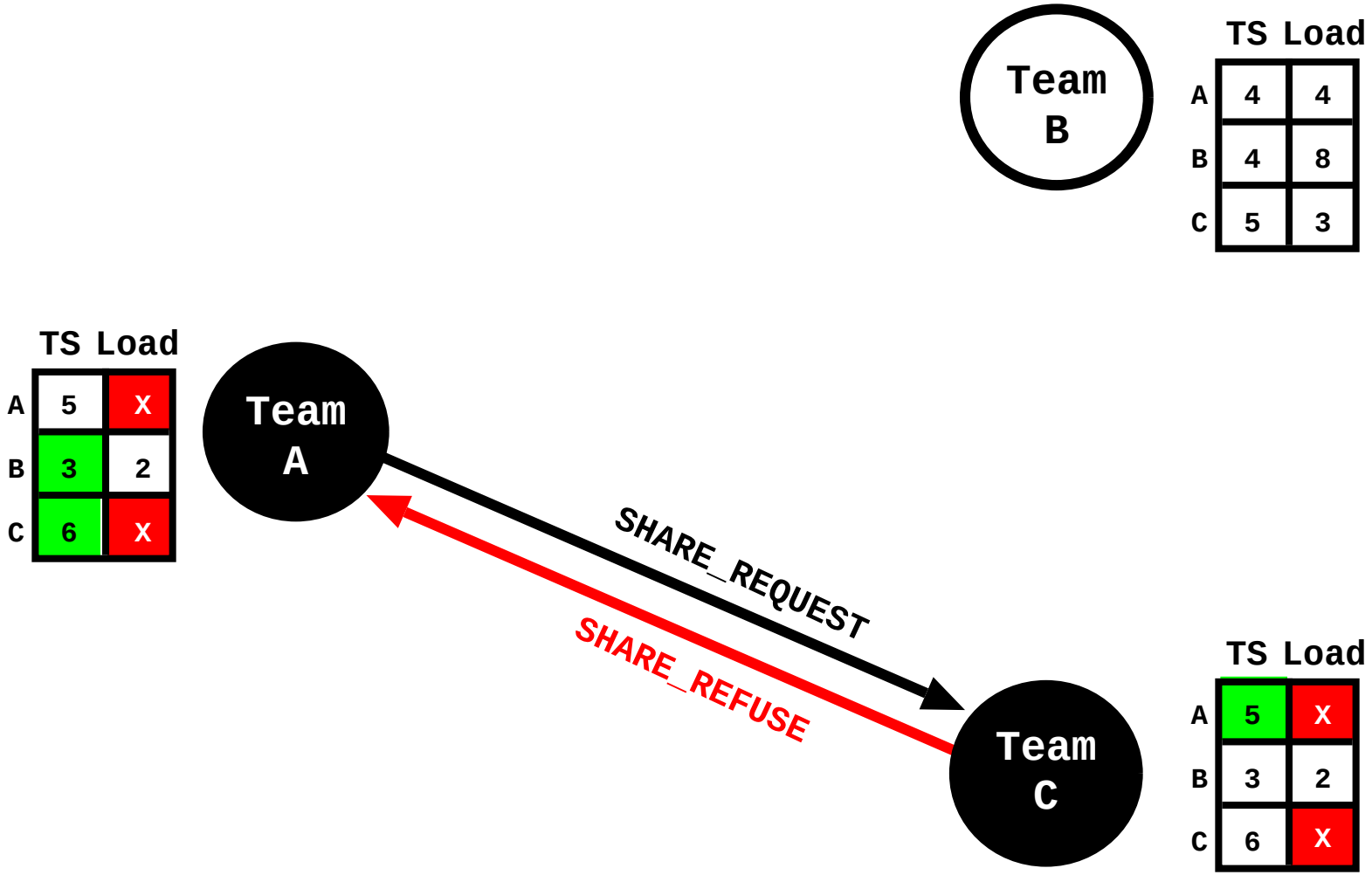




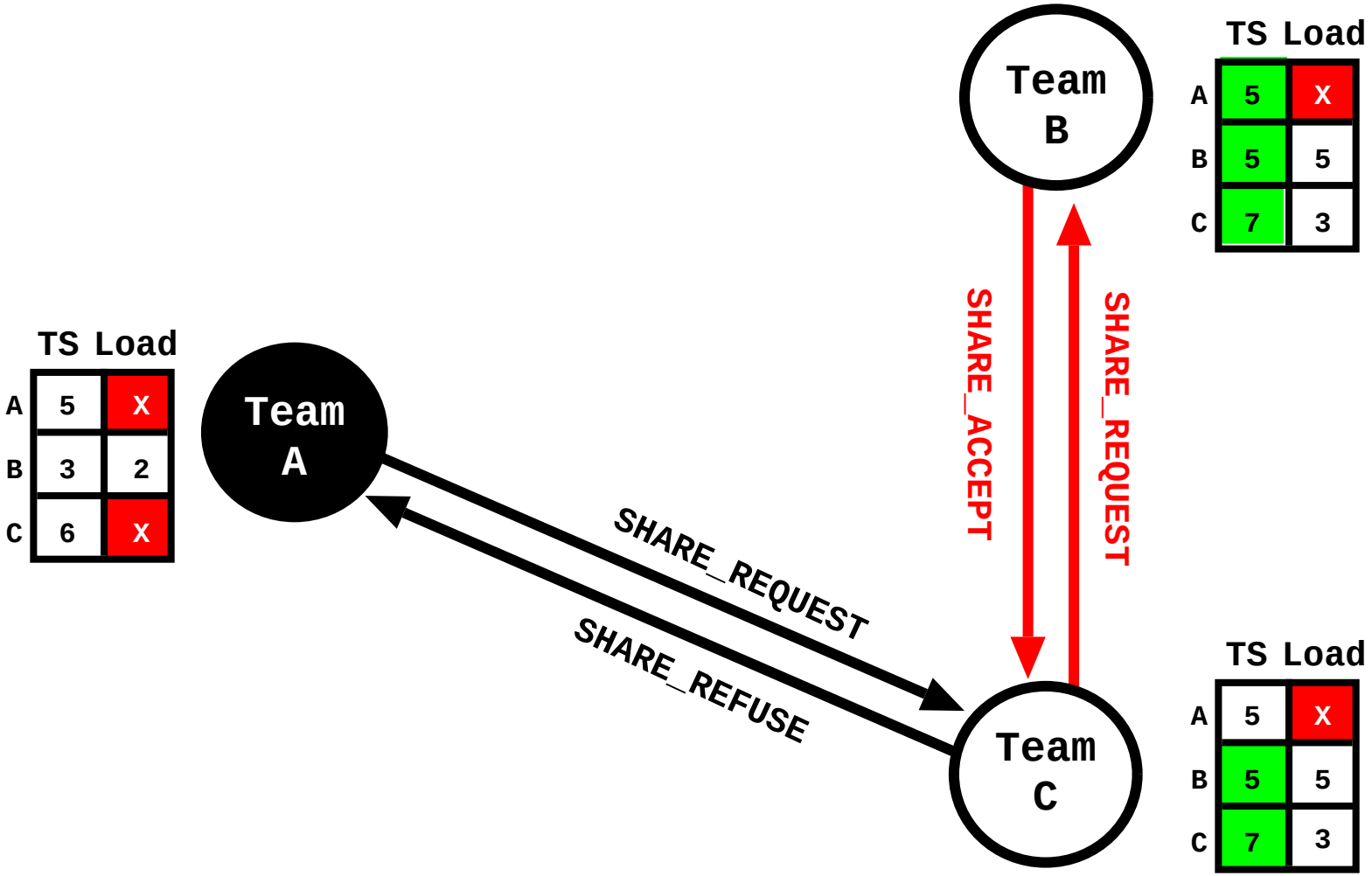
Termination With Timestamps



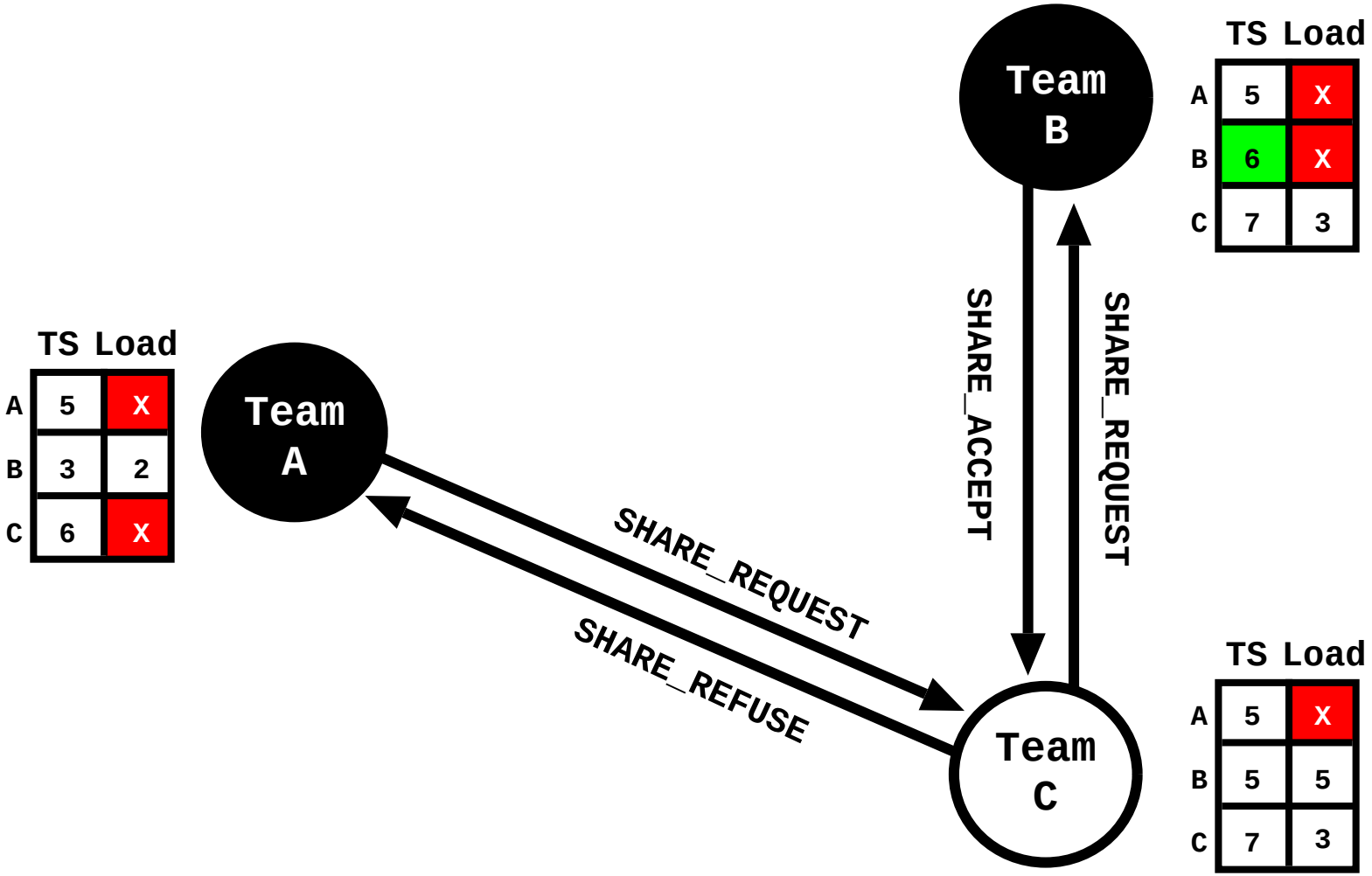
Termination With Timestamps



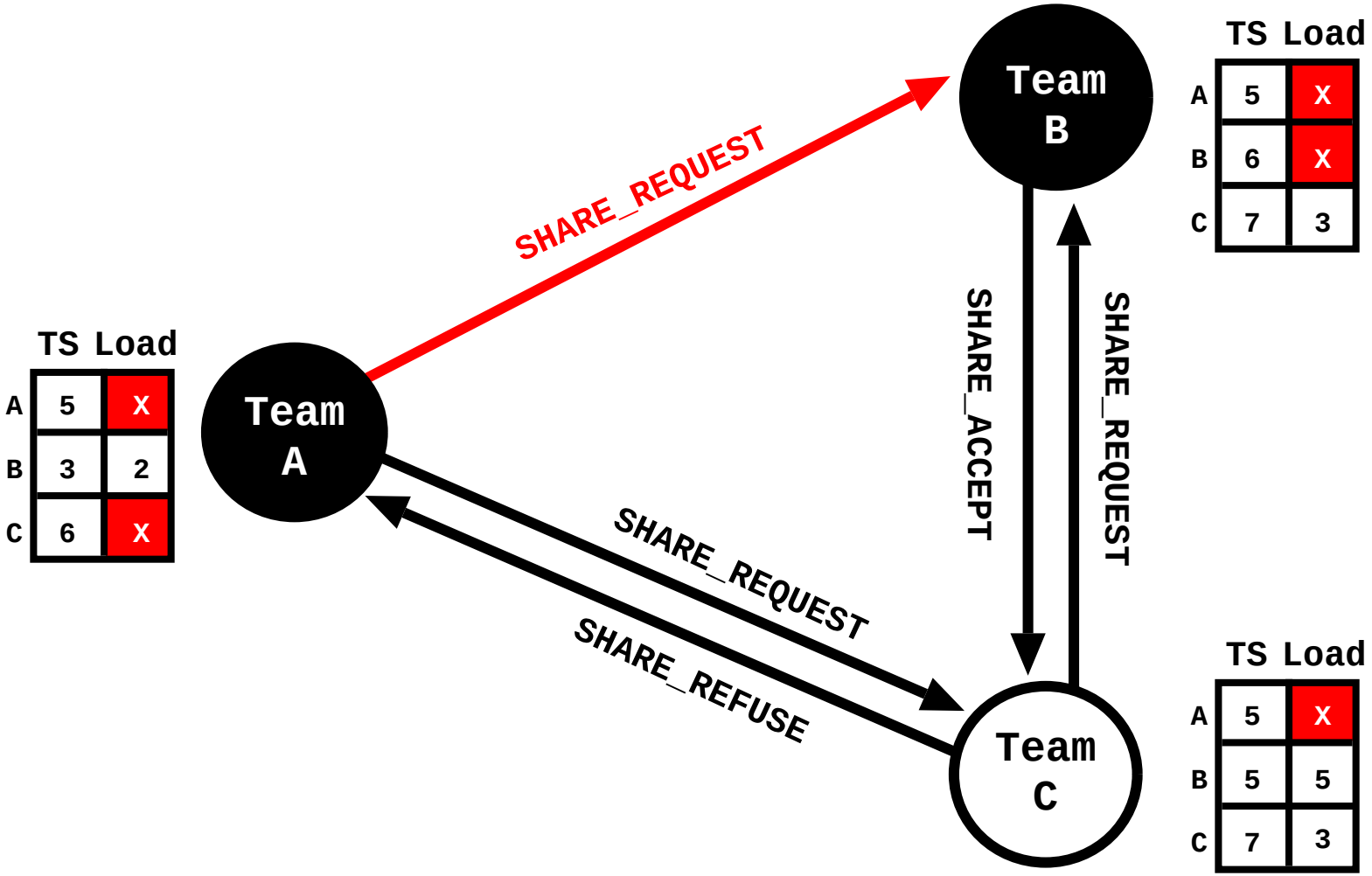
Termination With Timestamps



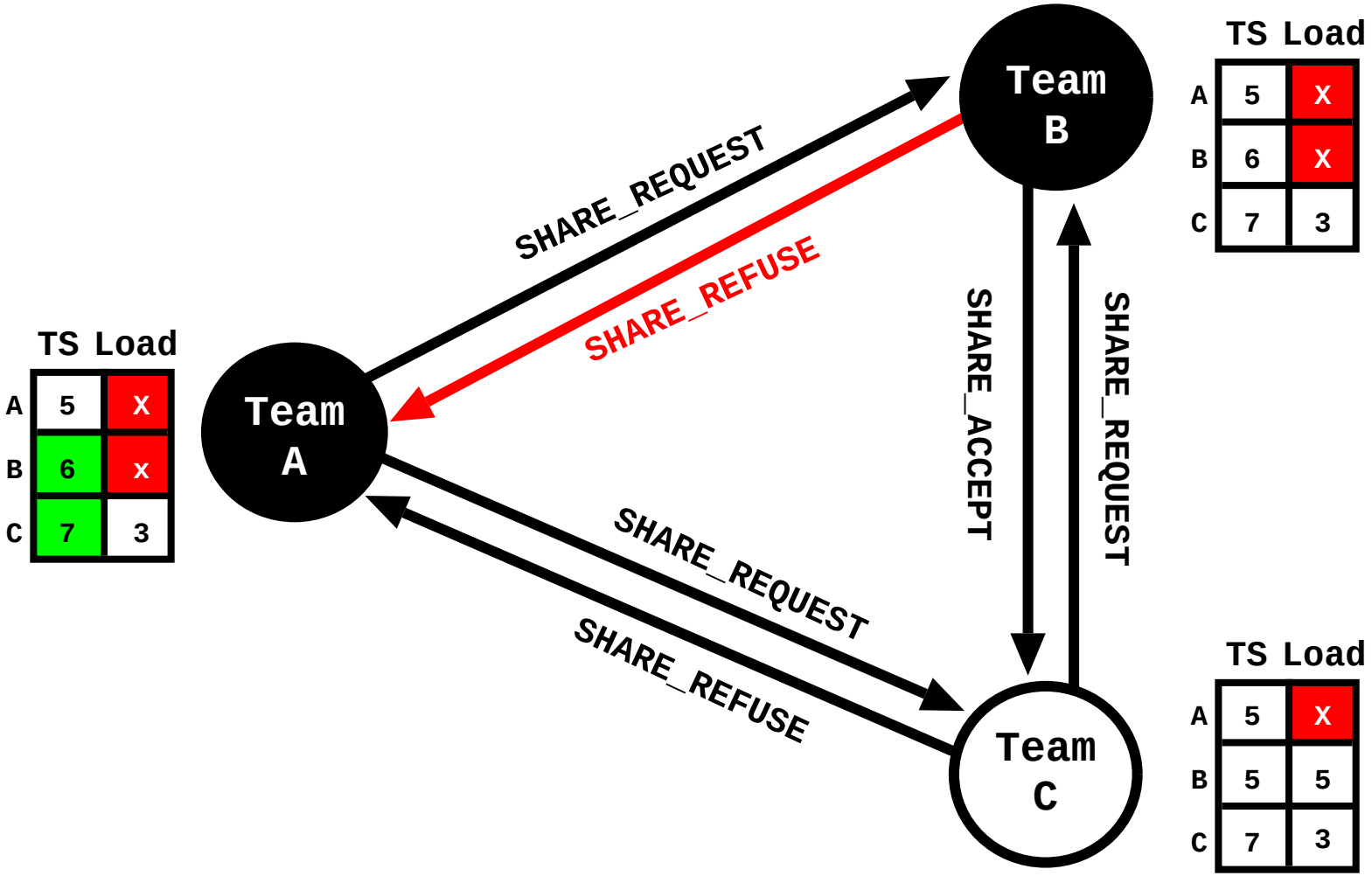
Termination With Timestamps



Termination With Timestamps



Termination With Timestamps



Performance Evaluation

- **2 parallel machines**, each one with 4 AMD SixCore Opteron TM 8425 HE @ 2.1 GHz (**24 cores per machine, 48 cores in total**) and 64 GBytes of main memory each, running Fedora 20 with the Linux kernel 3.19.8-100 64 bits.
- **10 well-known benchmark programs**. All together, the 10 benchmarks take **around 1800 seconds (30 minutes) to run with YapOr with a single worker**. All results are the average of 10 runs.
- We assume that **each team runs in a different machine**:
  - Configured OpenMPI to use the loopback interface and the TCP protocol for all communications
  - Used *tc* command to add more 0.06 milliseconds latency in the loopback interface (to simulate the latency observed between the physical machines)

Performance Evaluation

16 Workers	16W	2T 8W		4T 4W		8T 2W		16T 1W	
	YapOr	VS	HS	VS	HS	VS	HS	VS	HS
Average	14.60	13.20	13.46	12.64	12.93	10.29	11.75	7.04	7.94

24 Workers	2T 12W		4T 6W		6T 4W		12T 2W		16T 1W	
	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
Average	18.84	19.08	18.16	18.48	16.90	17.56	12.15	14.35	7.59	8.37

32 Workers	2T 16W		4T 8W		8T 4W		16T 2W		32T 1W	
	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
Average	24.10	23.94	22.98	23.35	20.26	21.40	13.08	15.35	7.70	8.86

Speedup results against YapOr execution with a single worker for different configurations of teams using vertical (VS) and horizontal (HS)

Performance Evaluation

16 Workers	16W	2T 8W		4T 4W		8T 2W		16T 1W	
	YapOr	VS	HS	VS	HS	VS	HS	VS	HS
Average	14.60	13.20	13.46	12.64	12.93	10.29	11.75	7.04	7.94

24 Workers	2T 12W		4T 6W		6T 4W		12T 2W		16T 1W	
	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
Average	18.84	19.08	18.16	18.48	16.90	17.56	12.15	14.35	7.59	8.37

32 Workers	2T 16W		4T 8W		8T 4W		16T 2W		32T 1W	
	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
Average	24.10	23.94	22.98	23.35	20.26	21.40	13.08	15.35	7.70	8.86

- #1: HS achieves better speedups than VS and the difference seems to increase as we increase the number of teams (probably as a result of more stack splitting operations)

Performance Evaluation

16 Workers	16W	2T 8W		4T 4W		8T 2W		16T 1W	
	YapOr	VS	HS	VS	HS	VS	HS	VS	HS
Average	14.60	13.20	13.46	12.64	12.93	10.29	11.75	7.04	7.94

24 Workers	2T 12W		4T 6W		6T 4W		12T 2W		16T 1W	
	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
Average	18.84	19.08	18.16	18.48	16.90	17.56	12.15	14.35	7.59	8.37

32 Workers	2T 16W		4T 8W		8T 4W		16T 2W		32T 1W	
	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
Average	24.10	23.94	22.98	23.35	20.26	21.40	13.08	15.35	7.70	8.86

- #2: fixing the total number of workers, speedups increase as we increase the number of workers per team (thus taking advantage of the maximum number of cores in a machine)

Performance Evaluation

16 Workers	16W	2T 8W		4T 4W		8T 2W		16T 1W	
	YapOr	VS	HS	VS	HS	VS	HS	VS	HS
Average	14.60	13.20	13.46	12.64	12.93	10.29	11.75	7.04	7.94

24 Workers	2T 12W		4T 6W		6T 4W		12T 2W		16T 1W	
	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
Average	18.84	19.08	18.16	18.48	16.90	17.56	12.15	14.35	7.59	8.37

32 Workers	2T 16W		4T 8W		8T 4W		16T 2W		32T 1W	
	VS	HS	VS	HS	VS	HS	VS	HS	VS	HS
Average	24.10	23.94	22.98	23.35	20.26	21.40	13.08	15.35	7.70	8.86

- #3: fixing the number of workers per team, speedups increase as we increase the number of teams (thus taking advantage of adding more computer nodes to a cluster)

Main Contributions and Further Work

- **Main Contributions**
  - Implementation of the layered model
  - New syntax
  - Performance study
- **Further Work**
  - Support incremental copy between teams
  - Support dynamic teams
  - Support dynamic code compilation
  - Support cut semantics
  - Avoid speculative work (new scheduling strategies)
  - Integrate with other YapOr extensions (ThOr, stack splitting)
  - ...

Thank You!

Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences
University of Porto, Portugal

ricroc@dcc.fc.up.pt
http://www.dcc.fc.up.pt/∼ricroc