Assignment 4

Distributed Runtime Environment

Parser classes

Instruction

Field	Type	Description
`tag`	`InstructionType`	Type of the instruction
`args`	`List[str]`	Arguments of the instruction
`label`	`Optional[str]`	Optional label for branching

Program

Field	Type	Description
`name`	`str`	Name of the program
`instructions`	`List[Instruction]`	All the instructions of the program
`labels`	`Dict[str, int]`	Map of label names to index of instruction

Parser

- Takes as input a string for the source code
- Checks for the first line to be “#SIMPLESCRIPT”
- Finds each instruction and checks against the valid tag and argument count.
- Stores the labels it finds along with the instruction index to which the label points
- Returns a Program object with the program name, instruction list and label dict

Interpreter and scheduler classes

ThreadState

Field	Type	Description
`thread_id`	`str`	ID of the thread (UUID)
`program`	`Program`	The program of the thread
`program_counter`	`int`	Index of current instruction
`variables`	`Dict[str, str | int]`	Map of variable name to value
`wakeup_timestamp`	`float`	The timestamp when the sleep expires
`tuple_pattern`	`List[str|int|Wildcard]`	The tuple pattern the thread is waiting for
`tuple_in`	`bool`	Request is IN or RD
`is_terminated`	`bool`	If the thread is terminated

TupleRequest

Field	Type	Description
`thread_id`	`str`	ID of the thread (UUID)
`owner`	`str`	Owner's node UUID
`completer`	`str`	Completer's node UUID
`pattern`	`List[str|int|Wildcard]`	Pattern of the tuple to complete the request
`is_destructive`	`bool`	True if IN or false if RD

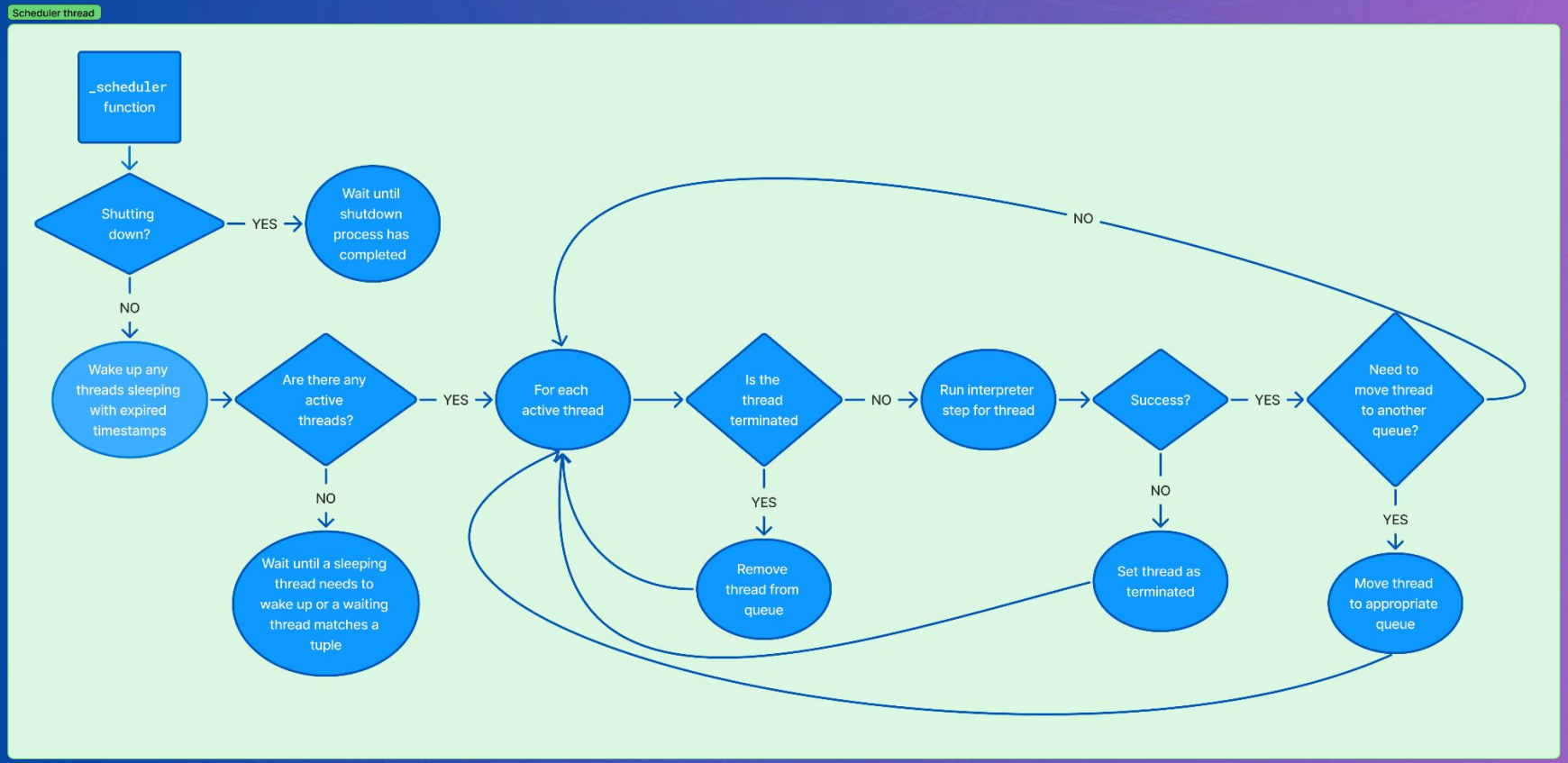
Interpreter

- Takes as input a ThreadState object and tries to step it forward by one instruction
- In case of an error it returns false to the upper layer, or else it returns true
- It uses the special fields in the thread state object: `wakeup_timestamp` and `tuple_pattern` if the next instruction is a sleep or a in/rd command respectively.
- Calls tuple space manager's functions if the command is a tuple command.

Scheduler

- Stores `ThreadState` objects in 3 different queues: `active_threads`, `sleeping_threads`, `waiting_threads`, based on if a thread has useful instructions to execute, has called sleep and is waiting for time to pass or has requested a tuple that has not been satisfied yet.
- Provides functionality for: Adding a thread, killing a thread, migrating a thread and load balancing the local threads.
- Has a separate thread that runs the main scheduling logic

Scheduler - Thread



Local tuple space manager

- Stores RD and IN commands as TupleRequest objects as a list. If the can be satisfied with the contents of the tuple space, they are completed immediately.
- OUT commands first try to complete as many pending requests as possible (All RD and at most one IN commands).
- If no IN requests were completed it stores the tuple in list of tuples.
- Provides option to forcefully add requests to the tuple space.

TupleRequest

Field	Type	Description
`thread_id`	`str`	ID of the thread (UUID)
`owner`	`str`	Owner's node UUID
`completer`	`str`	Completer's node UUID
`pattern`	`List[str|int|Wildcard]`	Pattern of the tuple to complete the request
`is_destructive`	`bool`	True if IN or false if RD

Distributed tuple space - Protocol

TupleInMessage

Field	Type	Description
`req`	`TupleRequest`	The request of a tuple

TupleConsumeMessage

Field	Type	Description
`thread_id`	`str`	Thread involved in the request

TupleOutMessage

Field	Type	Description
`node_id`	`str`	Completer
`thread_id`	`str`	Thread asking for the tuple
`tuple`	`tuple`	The proposed tuple

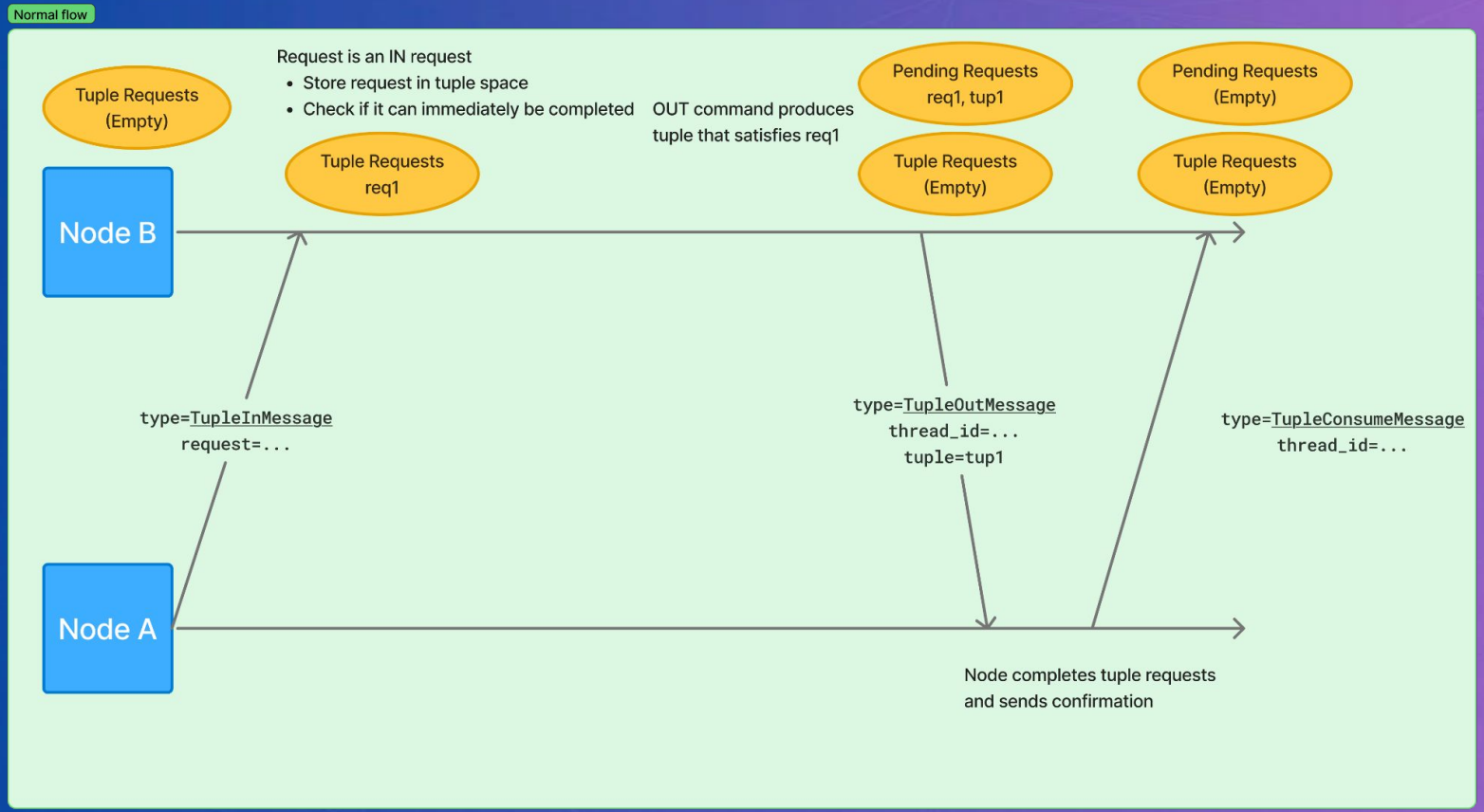
TupleCancelMessage

Field	Type	Description
`thread_id`	`str`	Thread involved in the request

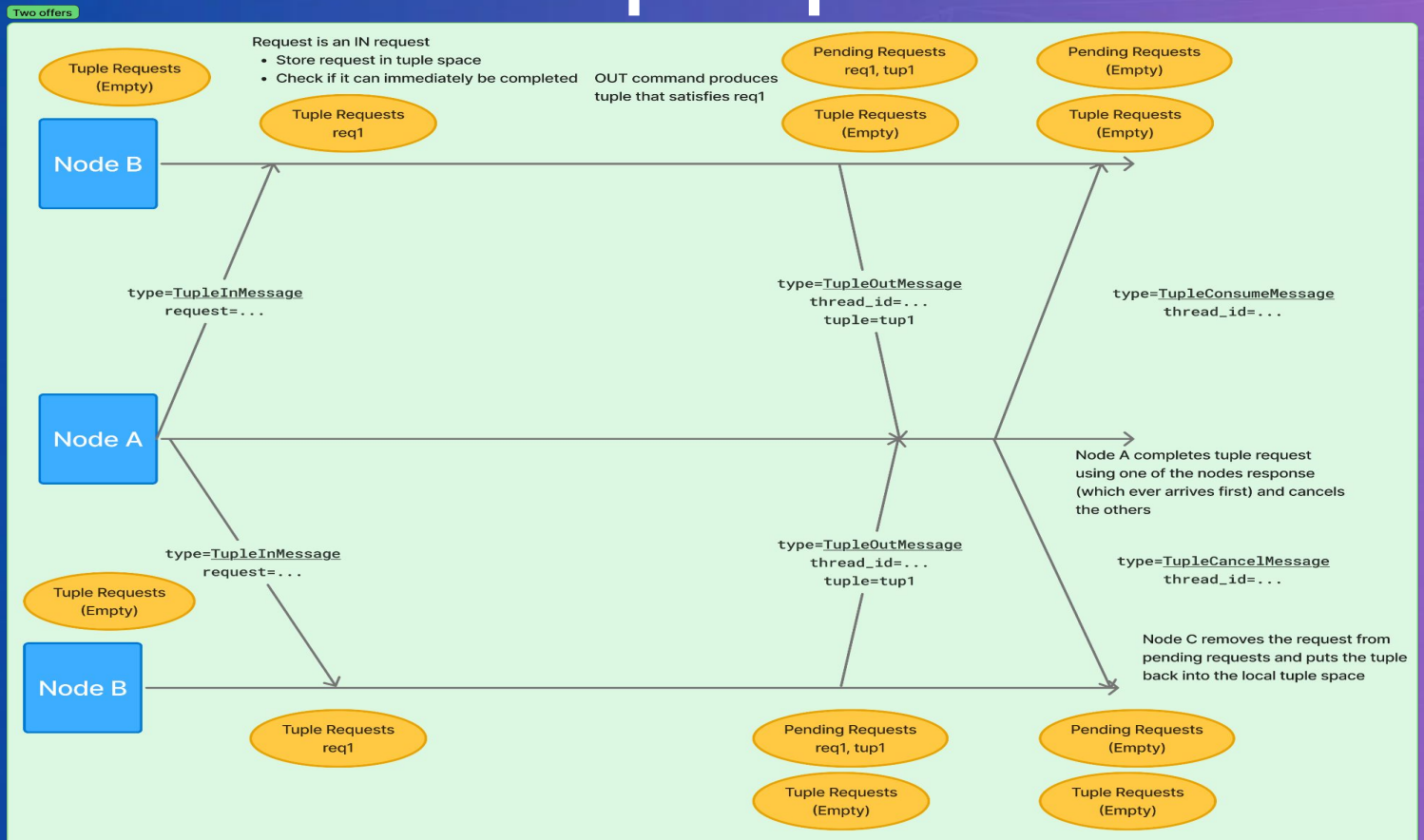
Distributed tuple space - Protocol

- For each IN or RD command, a broadcast `TupleInMessage` is sent to all the nodes. Each node receiving such message checks if it can complete it or stores it as a tuple request in the local tuple space.
- When a `TupleInMessage` is completed a `TupleOutMessage` is sent to the node that requested the tuple and if the request was an IN request, the “completer” node temporarily removes the tuple from the local space and waits for confirmation.
- The owner node of the `TupleInMessage` can respond with either a `TupleCancelMessage` or a `TupleConsumeMessage`.

Distributed tuple space - Protocol



Distributed tuple space - Protocol



Thread migration - Messages

ThreadMigrateMessage

Field	Type	Description
`node_id`	`str`	Sender of the thread
`thread`	`ThreadState`	The migrated thread

Thread migration - Protocol

- When a thread needs to get migrated, the scheduler initiates the process. It removes the thread for any queue it previously was.
- If the thread was in the waiting queue a `TupleCancelMessage` is broadcasted to every node and the tuple request is basically cancelled. The program counter is also decremented by one so the new node that gets the thread reruns the last instruction.
- In the case the thread was in the sleeping queue the `wakeup_timestamp` is the way the new node knows how much remaining sleep this thread has.

Load balancing - Protocol

- Broadcast a `LoadStatusMessage` to every other node.
- Check against your local table of node loads and find the least loaded node based on thread count.
- If the node has a load difference of 2 or larger, migrate a thread to it. This is to avoid unnecessary oscillations between nodes.
- Rerun this logic, periodically.

LoadStatusMessage

Field	Type	Description
`node_id`	`str`	ID of the sender
`load`	`int`	Thread count of the node

Shutdown - Messages

ShutdownPrepareMessage

Field	Type	Description
node_id	str	Node starting shut down

ShutdownPrepareAckMessage

Field	Type	Description
node_id	str	Node acknowledging the shutdown

ShutdownMessage

Field	Type	Description
node_id	str	Node acknowledging the shutdown
tuples	List[tuples]	Local tuple space of the node

Shutdown - Protocol

- When a node wants to shutdown and there are other nodes available, it first disables the scheduler, it broadcasts a `ShutdownPrepareMessage` to every other node.
- From now on, the node completes any other messages it receives and waits for a `ShutdownAckMessage` from every node.
- Each node receiving a `ShutdownPrepareMessage` removes the node from its table and sends a `ShutdownAckMessage`.
- After receiving all ACKs, the node can safely shutdown. It migrates all the threads and tuples to an available node and sends `ShutdownMessage` to all nodes.