Nicholas Locklear

G01090642

Lab 1 Design Document

Concurrent Traffic Control System for a One-Lane Bridge

This document outlines my design for a concurrent traffic control system over a one lane bridge, written in the language Go. Synchronization mechanisms such as mutex locks and condition variables are used to ensure safe access to shared resources while preventing race conditions and deadlocking. An assumption regarding the maximum number of groups was made. I assumed it was 2 groups at most, based on the information and test cases provided in the lab document. I apologize if this causes issues with testing or is otherwise incorrect.

So, let’s start with a high-level overview of the program, targeting the arrive, cross, leave, main, and the canCross methods. Assume for the purposes of this overview that atomicity is preserved whenever shared data is accessed or referenced.

When a vehicle arrives at the bridge, it first acquires a lock to ensure exclusive access to shared state variables. It updates the count of waiting vehicles based on its direction. The vehicle waits until the bridge is safe for it to cross, which is determined by the canCross function. If crossing is not allowed, the vehicle waits on the condition variable. Once allowed, it updates the bridge’s state by increasing the weight, updating the number of cars currently on the bridge, and updating the number of consecutive vehicles crossing in the same direction.

While crossing (in the cross function), the vehicles all are put to sleep for a period of 2 seconds to simulate travel time. It’s really simple, literally one line.

After crossing, the vehicle enters the leave function and acquires the lock again to update the bridge’s state. It reduces the bridge’s total weight and the vehicle count. If the bridge is now empty or too many vehicles have crossed in the same direction consecutively, it may switch the allowed direction to maintain fairness if it detects that the other direction has waiting vehicles. Finally, it signals the next waiting vehicle, ensuring progress in the system.

CanCross checks whether a vehicle is allowed to cross based on the bridge’s weight limit and direction rules. A vehicle can cross if adding its weight does not exceed the limit and if either the bridge is empty, or it is continuing in the current direction within the consecutive crossing limit. If too many vehicles have crossed in one direction, priority may shift to the opposite direction.  
  
And now, we will get into the data structures and synchronization techniques used, starting with the biggest part, the bridge itself.  
  
**Bridge Struct**

The Bridge struct represents the bridge and tracks its current state. It includes the following key fields:

* **mutex sync.Mutex** – A mutual exclusion lock that ensures only one goroutine modifies the bridge's shared state at a time, preventing race conditions.
* **cond \*sync.Cond** – A condition variable that allows vehicles to wait efficiently when they cannot cross yet. It works with the mutex to avoid busy waiting.
* **currentWeight int** – Tracks the total weight of vehicles currently on the bridge, ensuring that the bridge does not exceed its weight limit.
* **currentDir int** – Stores the direction of the vehicles currently crossing the bridge (Northbound or Southbound). If no vehicles are on the bridge, it is set to 0.
* **numVehicles int** – Keeps track of the number of vehicles currently on the bridge.
* **dirCount int** – Tracks how many consecutive vehicles have crossed in the same direction, enforcing the fairness constraint (max 6 consecutive vehicles).
* **waitingNorth int / waitingSouth int** – Counts the number of vehicles waiting to cross in each direction, helping determine when to switch directions.

**Synchronization Mechanisms**

Since multiple goroutines interact with shared bridge state, proper synchronization is required to prevent data races and ensure vehicles follow the crossing rules correctly.

1. **Mutex (sync.Mutex)**
   * Every vehicle must acquire the mutex lock before modifying shared bridge state (e.g., updating weight, direction, or vehicle count).
   * The mutex prevents simultaneous modifications from different goroutines, ensuring consistency.
2. **Condition Variable (sync.Cond)**
   * When a vehicle arrives and finds that it cannot cross, it waits on the condition variable (cond.Wait()).
   * Once conditions change (e.g., another vehicle leaves, reducing the weight), the bridge signals waiting vehicles using cond.Signal(), allowing them to check again if they can cross.
   * The use of Signal() (instead of Broadcast()) ensures that only one waiting vehicle is woken up at a time, reducing unnecessary wake-ups and potential congestion.
3. **Fairness Mechanisms**
   * The dirCount variable ensures that no more than 6 consecutive vehicles cross in the same direction before switching, preventing starvation.
   * The bridge direction (currentDir) switches when the bridge is empty or after 6 consecutive vehicles, ensuring a fair distribution of crossing opportunities.