Project Report: PlusCAL Ticketing System

João Kennedy RODRIGUES LEITÃO FILHO
Ana Julia MENDES GOUVEIA DA SILVA

Part 1: Verifying a honest system

To start, we worked with the base code that was given to us, following the established rules by the instructions, and taking some liberties with the rules that were not specified. For this first part, we didn't follow a very clear path like logic → implementation → bug fixes. We chosed this way because the code was very segmented, so we implemented features gradually. The Honest Client was implemented with the following logic:

- It has a variable called "current_seat" that indicates which seat it's looking forward to buying. To avoid that the client tries to buy the same seat forever, this variable starts at 1 and gets increased by 1 after each buy attempt, independent of being accepted or denied.
- The client may decide to cancel and be reimboursed for a ticket that he bought, but it follows a rule: it can only cancel the last ticket that he bought.
- The client stops when one of the following items is satisfied ("done" state):
    1. It reached the desired number of tickets.
    2. His bank account reaches zero.
    3. He went through all the seats twice.
- We included two safety invariants:
    - TypeOK: it checks if all the variables keep their expected type (like seatState can only be "available" or "paid" in this first version of the model).
    - MoneyConservation: It checks if the Honest Client has the same value of money during the execution. The start money must be equal to the number of tickets he thinks he has + remaining money in the end.
- Finally, we still included a liveness proprietary, which specifies that at some point (eventually) all the Honest Clients must reach the "done" state.

We had a problem with the Money Concervation invariant, so for this reason we kept it deactivated until we finished all the parts. The problem was caused by a moment during the server processing that would remove the money from the client account, but the ticket was not still assigned to him. In this way, the invariant would fail. We fixed it by making it dynamic, it will only check the conservation of money if there are no messages in the queue.

Part 2: Introducing a malicious client

For the second part, we followed a more strict step-by-step implementation. First, we started by spending some time analyzing the model and looking for potencial breaches to be exploited in our model. This way, we saw that it was not necessary to change the server or the Honest client logic, just adding a new one. So, we delimited the logic into the code, like where each part of the code would be insired, and checked again if it would work. After this we implemented the logic by translating the code into PlusCal and solved a bug that we found.

The Malicious client was implemented in a way that he does not try to buy any seat (that's the reason he does not start with any money), he has one cycle: Malicious client awaits that servers returns at least one seat as "paid" (public information). Then he sends a cancellation message to the main server, using the target seat ID, and instead of the real owner's bank account to receive the reimbursed money, he sends his bank account ID, and this surprisingly works.

The bug that we faced was caused because we forgot to add a rule that he (Malicious client) has to wait for the last cancellation scam result to try to act again. In this way, he was able to make several cancellation requests even before receiving the result for the first request. This was fixed by adding a "wait for server's answer" rule.

For this part, the invariant "Money Conservation" is not violated because the Malicious entity performed a stealth scam. In the way it was implemented it`s focused on the point of view of the client, and not the ground-truth, so each client thinks they still have the ticket and are fine with their Money Conservation. This proves how dangerous it is to keep an unsecure system.

Part 3: Fixing and extending the protocol

3.1 Security

We started part 3 by fixing the problem, it was quite easy since we just needed some lines of code to assign the ownership of a seat to an Honest Client, so we skipped the "logic thinking" part and went straight into the implementation.

For it, we introduced a variable called "seatOwner" that tracks the information of which client owns which seat. This variable can only be modified by the server. In this way, a cancellation can only be processed if the seat ownership is the same as the ID that sent the message.

Here we faced a bug which the Malicious client never gave up, even if the server was denying every single request. To solve this problem we implemented a variable to the Malicious client called “rounds” (same as Honest Clients), that if his money does not change after 2 scam attempts, he will give up and finish.

3.2 Extending the protocol

In the final part of the project, our goal was to make the system more realistic, efficient, and functional. We introduced several new features, including support for seat cancellations and the reservation of multiple seats. We added an extra step to request the list of available seats to reduce the chances of selecting seats that are already taken. We also included a reservation phase before payment and allowed users to change their IP address without compromising security.

For the first feature, we updated the data model to accept sets of seats as input instead of single integers. We adapted the rest of the code accordingly. This required changes to both the message definitions and the server logic to ensure proper handling of atomic transactions. The server now checks that all seats in a requested batch are available before confirming any operation. Additionally, we created a new reserved state in the seat map. This establishes a two-step process (Reserve, then Buy) that prevents race conditions from affecting the payment process.

We also upgraded the Honest client, making him a "Smart Client." Previously, he guessed seat numbers without much success. Now, he follows a clear protocol: if he doesn’t know which seats are free, he sends a query message. The server responds with the complete list of available seats, and the client selects a subset from that list to reserve. This change prevents him from asking for seats that are already taken.

To meet the requirement that allows users to change their IP address without losing their tickets, we realised it was a design flaw to use the IP address as the only user ID. We fixed this by separating identity from connectivity. We added a senderID field to the messages. Now, the server checks permissions (like who owns a seat for cancellation) using the persistent senderID, but replies back to the current IP. This lets the client update his IP address during the session without getting locked out of his account.