Distributed Stable Marriage with Java RMI

Panayiotis Charalambous - Marios Stavrou

20 Feb 2026

Problem Statement

We implement a distributed peer-to-peer application that solves the *Stable Marriage Problem* for n men and n women. Each man and woman has a strict preference ordering (a permutation) over the opposite set. Men propose to women; women accept the first proposal they receive, but may later *dump* their current partner if a better proposal arrives. The goal is to reach a one-to-one matching with no blocking pairs, i.e., a stable matching.

Approach and Distributed Model

We model each `Man` and `Woman` as a separate RMI-exported object. Communication happens via message passing implemented as Java RMI calls:

- **Proposal:** `Man` calls `Woman.onProposal(manId)`.
- **Response:** `Woman` returns `ACCEPT` or `REJECT` to the proposer.
- **Dump notification:** if a woman switches partners, she notifies the previous partner using `oldMan.onResponse("DUMP", womanId)`.

Women maintain a *rank* array for $O(1)$ comparison between the current partner and a new proposer. Each man maintains a pointer into his preference list and repeatedly proposes to the next woman upon rejection or dump. A trace is printed for key events (propose/accept/reject/dump/engage) to observe progress.

Concurrency and Correctness Notes

RMI may dispatch multiple incoming calls concurrently, so women must protect shared state (e.g., `partnerId`) against races. We synchronize the critical region inside `Woman.onProposal` to ensure that accept/dump decisions are atomic. To avoid deadlocks in a recursive proposal style, we ensure that *remote calls are not performed while holding the woman lock*: the decision (including which previous partner to dump) is computed inside the synchronized block, and the dump notification is sent *after* the lock is released.

Termination Condition

In practice, we terminate the simulation when all men are engaged (each man has `partnerId` set). This is a simple and effective condition for the simulation driver and aligns with the algorithm reaching a final matching.

Build Issue Encountered (Stale Class Files)

During testing, we observed intermittent `ArrayIndexOutOfBoundsException` errors after changing n (e.g., from 10 to 12). The root cause was stale compiled `.class` files: the runtime was sometimes using old bytecode compiled with a different constant N, leading to mismatched array sizes (e.g., `rank.length = 10` while receiving `manId = 11`). The solution was to enforce clean rebuilds (remove `.class` files before recompiling) to guarantee that all classes are compiled consistently with the updated N.

Makefile for Reliable Builds

To prevent stale build artifacts and simplify execution, we introduced a Makefile that:

- compiles all Java sources,
- runs the `Driver`,
- supports clean rebuilds (`make clean` / `make rebuild`),
- optionally kills any leftover process holding the RMI registry port (1099).

This made testing with different n values reliable and repeatable.