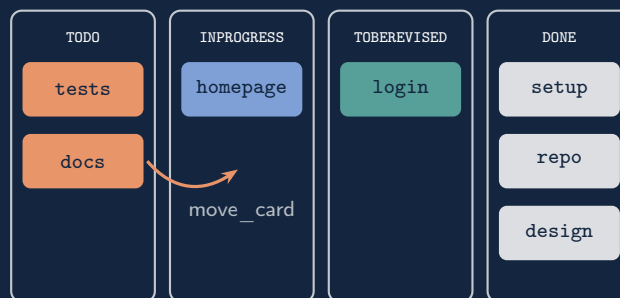




UNIVERSITÀ DI PISA

WORTH

WORKTogetHer • a Kanban board for distributed teams



TCP 45678 requests and responses

RMI 56789 registration, callbacks

UDP 13731 one multicast chat per project

Java • sockets • threads • NIO

WORTH: a Distributed, Kanban-Inspired Collaboration Platform in Pure Java

Project Report

Giuseppe Muschetta

B.Sc. in Computer Science

Course: Network Programming • Course Instructor: Prof. Laura Ricci

Abstract

WORTH (WORkTogetHer) is a client-server system for managing collaborative projects, inspired by the Kanban method. Registered users create projects, add members, put cards on a board of four lists and move them through their life cycle according to fixed rules, see who is online, and talk in a chat that belongs to each project. The system uses no framework: every interaction is built on the Java standard library, choosing for each one the mechanism that suits it. Requests and responses travel over TCP, as JSON objects preceded by their length; registration and server-to-client notifications use Java RMI and its callbacks; each project has its own group chat over UDP multicast, with addresses taken from a managed pool of 8,192; the whole state is saved to disk with NIO channels after every change and restored when the server restarts. The server is multithreaded, serves each client in a thread of a cached pool and keeps the shared state consistent with synchronized blocks; passwords are never stored, only a salted PBKDF2 hash. This report describes the architecture, the protocols, the server and the client, and the Kanban rules that the server enforces, and closes with a scripted session that exercises every request, the callbacks, the chat, persistence across a restart and an abrupt disconnection: all of them behaved as specified.

Contents

1	Introduction	2
2	Architecture	2
3	Communication	3
3.1	Requests and responses over TCP	3
3.2	Registration and callbacks with RMI	4
3.3	Project chats over UDP multicast	5
4	The server	5
4.1	Startup and persistence	5
4.2	Serving many clients	6
4.3	Consistency	6
4.4	Disconnections	6
5	The WORTH service	6
5.1	Users and sessions	6
5.2	Projects and members	6
5.3	Cards and the Kanban rules	6
6	The client	8
6.1	The command line	8
6.2	Threads of the client	8
7	Passwords	8
8	Verification	9
9	Building and running	10
10	Conclusion	10

1 Introduction

Kanban is a method for managing work that makes its progress visible: every activity is a card on a board, and the board is divided into lists that represent the phases of the work [1]. Moving a card from one list to the next records that the activity has progressed, and a glance at the board shows the state of the whole project. WORTH (WORKTogetHer) brings this idea to a team whose members work on different machines. It offers:

- **accounts:** users register with a nickname and a password and log in to the service;
- **projects:** a logged-in user creates projects and adds other registered users as members; only members can see and change a project;
- **cards:** each project has a Kanban board with four lists, **todo**, **inprogress**, **toberevised** and **done**; cards are added to **todo** and moved between the lists according to fixed rules, and each card keeps the history of its moves;
- **presence:** every client knows which users are registered and which are online, updated in real time;
- **chat:** every project has a group chat for its members, in which the service itself announces the relevant events, such as a card that changes list.

WORTH was the final project of the Network Programming course. Its main constraint is also its main interest: there is no framework, and every networking primitive comes from the Java standard library. Each kind of interaction therefore gets the mechanism that suits it best: reliable request-response over TCP, remote method invocation for registration and for notifications pushed by the server, IP multicast for group communication, and NIO channels for persistence. The only external library is Jackson, for the conversion between Java objects and JSON.

The project consists of 19 classes and about 3,600 lines of Java. Section 2 presents the architecture, Section 3 the three communication mechanisms, Section 4 the server and Section 5 the service it offers, with the Kanban rules. Section 6 describes the client, Section 7 the handling of passwords, and Section 8 the scripted session used to verify the system. Section 9 explains how to compile and run it.

2 Architecture

WORTH follows the client-server paradigm. A single server holds the state of the service: the registered users, the projects with their members and cards, and the pool of addresses for the chats. Any number of clients connect to it, each one used by one person through a command-line interface. Figure 1 shows the components and how they communicate.

The classes divide the work as follows (the list follows the class descriptions of the project):

- **shared model:** **User**, **Project**, **Card** and **Chat** describe the entities of the service; **Message** is the object exchanged over TCP, and the enumerations **Request** and **Response** list the kinds of request and of answer;
- **server:** **ServerMain** starts it; **ServerCore** accepts connections, owns the thread pool, implements the remote interface **ServerInterface**, performs the callbacks and saves the state; **ServerThread** is the task that serves one client; **WorthCore**, which implements **WorthInterface**, holds the state and the operations of the service;
- **client:** **ClientMain** starts it; **ClientMenu** reads and checks the user's commands; **ClientCore** carries them out by talking to the server and implements the remote interface **ClientInterface** used for the callbacks; **ChatSaver** is the task that listens to the chat of a project;
- **security:** **HashPassword** computes and verifies the salted password hashes.

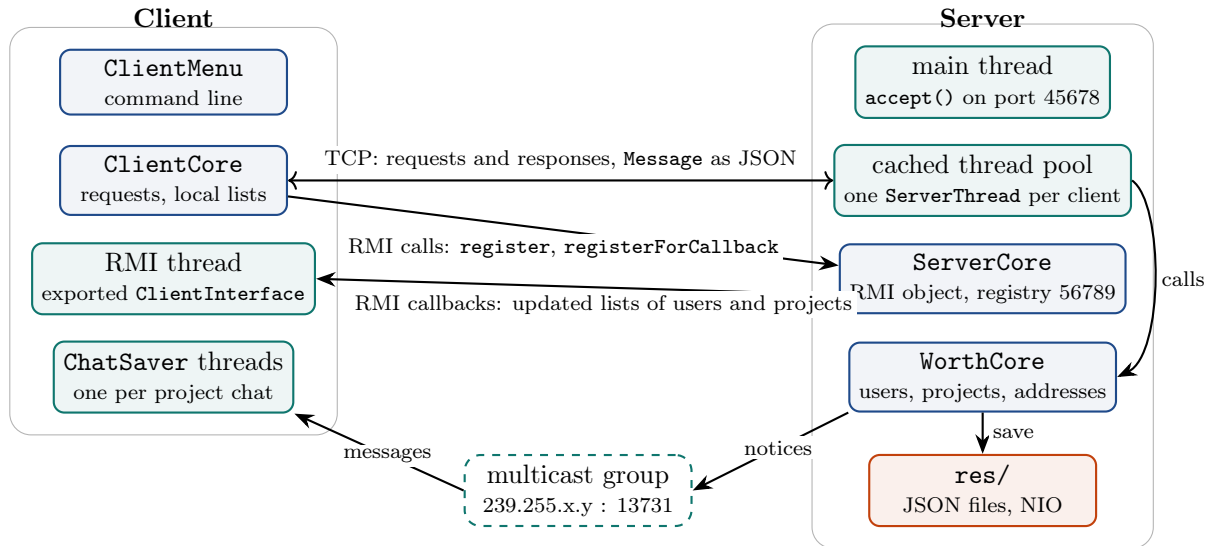


Figure 1: The architecture of WORTH. Blue boxes are classes, green boxes are threads, orange boxes are storage; every kind of arrow is a different communication mechanism. Members write to the chat group directly from their client.

3 Communication

WORTH uses three communication mechanisms, each for a different kind of interaction. Table 1 summarizes them.

Table 1: The communication mechanisms of WORTH.

Mechanism	Port	Used for
TCP	45678	login, logout and every operation on projects and cards: one request, one response
Java RMI	56789 (registry)	registration of new users (client to server) and callbacks with the updated lists of users and projects (server to client)
UDP multicast	13731	one group chat per project, for its members and for the service

3.1 Requests and responses over TCP

A client opens a TCP connection to the server when its user logs in, and closes it at logout. Over this connection the two sides exchange **Message** objects. A message carries the kind of request (an element of **Request**) and its arguments from the client, and the outcome (an element of **Response**) with the requested data from the server: the logged-in user, a list of projects, members or cards, or a single card. Table 2 lists the 11 requests with the outcomes that the server can return.

Framing. Messages are serialized to JSON with Jackson, and the JSON string is sent as bytes over the socket. Since a message can contain lists of any length, its size cannot be known in advance; for this reason the sender first writes the size as a 4-byte integer, then the bytes of the message (Figure 2). The receiver reads the integer, allocates an array of exactly that size and fills it with `readFully`, so it never reads too little or too much. The streams are wrapped in buffered streams, which group the bytes into few system calls.

Listing 1: Sending a response, from **ServerThread** (variable names translated, date settings omitted).

Table 2: The requests sent over TCP and their possible outcomes. Every request on a project also answers `NONEXISTENT_PROJECT` if the project does not exist or the caller is not one of its members.

Request	Outcomes besides OK
LOGIN	NOT_REGISTERED, WRONG_PASSWORD, ALREADY_LOGGED
LOGOUT	UNKNOWN_ERROR
LIST_ALL_PROJECTS	— (the projects of which the caller is a member)
CREATE_PROJECT	PROJECT_EXISTS, UNABLE_CREATE_PROJECT (no chat address left)
ADD_MEMBER	NOT_REGISTERED, MEMBER_EXISTS
SHOW_ALL_MEMBERS, SHOW_ALL_CARDS	—
SHOW_CARD	NONEXISTENT_CARD
ADD_CARD	CARD_EXISTS
MOVE_CARD	NONEXISTENT_LIST, NONEXISTENT_CARD, CARD_EXISTS (same list), MOVE_CARD_FORBIDDEN
CANCEL_PROJECT	DELETE_FORBIDDEN (some card is not done)

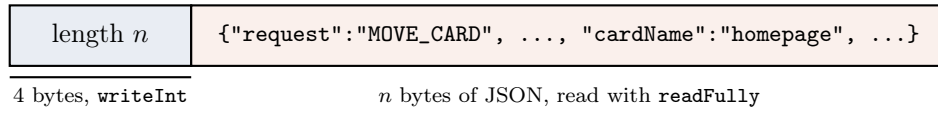


Figure 2: A message on the TCP stream: its length, then its JSON serialization.

```
private void sendToClient(Message message) throws IOException {
    DataOutputStream outStream = new DataOutputStream(
        new BufferedOutputStream(clientSocket.getOutputStream()));
    ObjectMapper mapper = new ObjectMapper();
    mapper.setVisibility(PropertyAccessor.FIELD, JsonAutoDetect.Visibility.ANY);
    byte[] bytes = mapper.writeValueAsString(message).getBytes();
    outStream.writeInt(bytes.length);    // the length first
    outStream.write(bytes);             // then the JSON bytes
    outStream.flush();
}
```

3.2 Registration and callbacks with RMI

Two interactions do not fit the request-response pattern of a TCP session. Registration happens before a user has a session at all, and notifications start from the server, when something changes that a client has not asked about. Both use Java RMI [2].

At startup the server exports itself as a remote object implementing `ServerInterface` and publishes its stub in an RMI registry, created on port 56789 under the name `SERVER-WORTH`. A client looks the stub up when it starts and calls its remote methods:

- `register(nickname, password)` creates a new user and answers `OK` or `USER_EXISTS`;
- `registerForCallback(stub)` and `unregisterForCallback(stub)` add and remove the client to and from the list of clients to be notified. A client registers right after a successful login, passing the stub of its own exported object, which implements `ClientInterface`.

The server calls back the clients through two remote methods of `ClientInterface`: `notifyUserEvent` delivers the updated list of registered users, with their online status, and `notifyChatsEvent` delivers the list of projects, from which each client extracts the chats of the projects it belongs to. The server sends these updates whenever the corresponding state changes: a registration, a login or a logout for the users; the creation or deletion of a project or the addition of a member for the chats. Since a callback is a remote call, it runs on an RMI thread of the client, concurrently with the user's commands, and the client protects its local

lists with synchronized blocks.

3.3 Project chats over UDP multicast

A chat is a group conversation: every message must reach every member, and the members change over time. IP multicast fits this exactly, since a datagram sent to a group address is delivered to every socket that joined that group. Each project receives its own group address when it is created, and all the chats use the same port, 13731.

The address pool. The addresses are taken from the range 239.255.224.0–239.255.255.255, inside the administratively scoped block 239.0.0.0/8, reserved for use within an organization [3]. The range contains $32 \times 256 = 8,192$ addresses, which is also the maximum number of projects that can exist at the same time. The server keeps the free addresses in a `TreeSet` whose comparator pads every octet to three characters before comparing the strings, so that the lexicographic order of the padded strings is the numeric order of the addresses. A new project takes the smallest free address with `pollFirst()`, and a deleted project gives its address back to the set; if the set is empty, the creation of a project answers `UNABLE_CREATE_PROJECT`.

Sending and receiving. A message is sent as one datagram through an ordinary `DatagramSocket` addressed to the group of the chat. To receive, the client starts one thread per chat of its user, each running a `ChatSaver` task: the task joins the group with a `MulticastSocket` bound to the common port, and stores every message it receives in the list of the chat, from which the user reads them with the command `receive`. Sockets bound to the same port can coexist because the address-reuse option of multicast sockets is enabled by default. The task waits on `receive` with a timeout of 2 seconds, so that it regularly checks whether the user is still online and whether it has been interrupted: a thread is stopped when its project is deleted, and all of them when the user logs out. Besides the members, the server itself writes in the chats: it announces the creation of the project, every new member, every new card and every move of a card.

4 The server

4.1 Startup and persistence

The state of WORTH survives the server: it is stored in the folder `res` in the working directory of the server, with one file for the registered users and one folder per project:

<code>res/users.json</code>	every registered user, with the hash of the password
<code>res/<project>/members.json</code>	the nicknames of the members of the project
<code>res/<project>/<card>.json</code>	one file per card, with its description, list and history

At startup, `ServerCore.begin()` restores the state: it reads the users, then each project folder, putting every card back in the list recorded in its file, and assigns a chat address to each restored project. If the folder does not exist yet, it creates it. Every operation that changes the state saves the affected files immediately, so the files on disk always reflect the last completed operation. Files are written and read through NIO `FileChannels` and `ByteBuffers`, after converting the objects to JSON with Jackson; Listing 2 shows the file of a card after the session of Section 8.

Listing 2: The file `res/website/homepage.json`, written by the server.

```
{"name":"homepage","description":"Landing page with the project list",
  "history":"TODO -> INPROGRESS -> TOBEREVIEWED -> INPROGRESS -> DONE","position":"DONE"}
```

Once the state is restored, the server exports its remote object, creates the registry, and enters a loop in which it blocks on `accept()` on the listening socket, waiting for clients.

4.2 Serving many clients

Each accepted connection becomes a `ServerThread` task submitted to a *cached* thread pool. The pool creates a new thread when no idle one is available and reuses threads that have been idle for less than 60 seconds, so the main thread is always free to accept new connections while each client has a thread devoted to it. The task reads one request at a time, calls the corresponding method of `WorthCore` and sends back the `Message` that the method returns, until the client logs out with a `LOGOUT` request.

4.3 Consistency

Several `ServerThread` tasks can run the methods of `WorthCore` at the same time, and they all work on two shared lists: the registered users and the created projects. The operations that change these lists do so inside blocks synchronized on the list, and those in which a check must be atomic with the update that follows it perform both inside the same block. For example, `createProject` checks that no project with the same name exists and adds the new project within one synchronized block on the list of projects: two clients that create a project with the same name at the same moment cannot both succeed. The address pool is changed only through two synchronized methods, `bindChatAddress` and `unBindChatAddress`.

4.4 Disconnections

A client can disappear without logging out, for instance when its process is killed. The server detects it when the read of the next request fails with an `EOFException`: the task then logs the user out and closes the socket. A disconnected client also leaves a stale stub in the list of clients registered for callbacks; the next callback to that stub fails, and the server removes the stub from the list instead of stopping the notifications to the others.

5 The WORTH service

5.1 Users and sessions

A user is identified by a nickname, unique in the service. `login` checks, in order, that the nickname is registered, that the password matches the stored hash (Section 7) and that the user is not already logged in from another client, and then marks the user as online and sends back their data. `logout` marks the user as offline and notifies every client of the change.

5.2 Projects and members

The user who creates a project becomes its first member, and any member can add other registered users. A project is visible only to its members: every request about a project first checks that it exists and that the caller is a member, and otherwise answers `NONEXISTENT_PROJECT`, exactly as if the project did not exist. A project can be canceled only when all its cards are in `done`: its chat address goes back to the pool and its folder is deleted from `res`.

5.3 Cards and the Kanban rules

A card has a name, unique within its project, and a description. It is created in `todo` and moves according to the rules of Figure 3: forward from `todo` to `inprogress`, from `inprogress` to

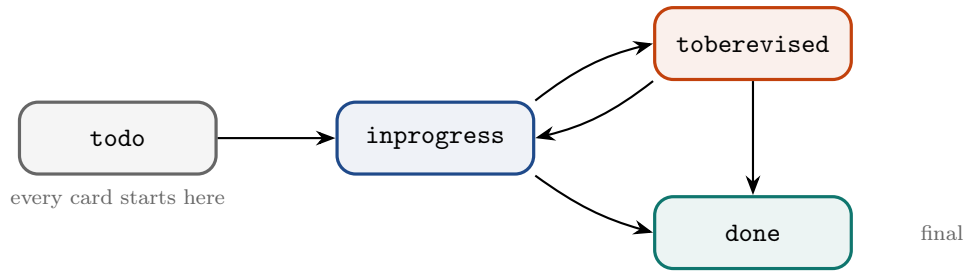


Figure 3: The moves allowed between the four lists; any other move is answered with `MOVE_CARD_FORBIDDEN`.

`toberevised` or `done`, and from `toberevised` back to `inprogress` or forward to `done`. Nothing can go back to `todo`, and `done` is final.

`moveCard` checks the request in a fixed order: that the project exists and the caller is a member, that both list names are valid (`NONEXISTENT_LIST`), that the two lists differ (`CARD_EXISTS`), that the move is allowed (`MOVE_CARD_FORBIDDEN`), and finally, inside a synchronized block, that the card is really in the source list (`NONEXISTENT_CARD`). Only then the card is moved, its history updated, the project saved and the move announced in the chat. Listing 3 shows the check of the rules.

Listing 3: The Kanban rules, from `WorthCore.moveCard` (comments added).

```

switch (destList) {
  case "todo" :
    // no card can go back to todo
    message.setResponse(Response.MOVE_CARD_FORBIDDEN);
    return message;
  case "inprogress" :
    if (sourceList.equals("done")) {
      // done is final
      message.setResponse(Response.MOVE_CARD_FORBIDDEN);
      return message;
    }
    break;
  case "toberevised" :
    if (!sourceList.equals("inprogress")) {
      // only work in progress is revised
      message.setResponse(Response.MOVE_CARD_FORBIDDEN);
      return message;
    }
    break;
  case "done" :
    if (sourceList.equals("todo")){
      // a card must be started first
      message.setResponse(Response.MOVE_CARD_FORBIDDEN);
      return message;
    }
    break;
  default :
}

```

Every move appends the destination list to the card's *history*, a string that starts with `TODO`: after a full cycle with one revision it reads `TODO -> INPROGRESS -> TOBEREvised -> INPROGRESS -> DONE`. The history and the current list are part of the card's file, so both survive a restart of the server.

6 The client

6.1 The command line

The user interacts with WORTH through a command-line interface. `ClientMenu` reads each line, checks the name of the command and the number of its arguments, and calls the corresponding method of `ClientCore`, which carries it out and returns the outcome to be shown. Only the commands that make sense at that moment are available: before login, `register`, `login`, `help` and `exit`; after login, the 17 commands of Table 3.

Table 3: The commands available after login, and how the client serves them.

Command	Arguments	Served by
<code>list_projects</code>	—	TCP request
<code>create_project</code> , <code>show_members</code> , <code>show_cards</code> , <code>cancel_project</code>	<code><project></code>	TCP request
<code>add_member</code>	<code><project></code> <code><nickname></code>	TCP request
<code>show_card</code> , <code>get_card_history</code>	<code><project></code> <code><card></code>	TCP request
<code>add_card</code>	<code><project></code> <code><card></code> <code><description></code>	TCP request
<code>move_card</code>	<code><project></code> <code><card></code> <code><source></code> <code><destination></code>	TCP request
<code>logout</code>	<code><nickname></code>	TCP request
<code>list_users</code> , <code>list_online_users</code>	—	local lists
<code>send</code>	<code><project></code> <code><message></code>	UDP multicast
<code>receive</code>	<code><project></code>	local chat list
<code>help</code> , <code>exit</code>	—	local

Four commands never reach the server through TCP. The lists of users come from the local copies that the callbacks keep up to date, so `list_users` and `list_online_users` answer immediately. `send` writes directly to the multicast group of the project, and `receive` shows the messages collected so far by the `ChatSaver` of that chat.

6.2 Threads of the client

A running client has three kinds of threads:

- the **main thread**, started by `ClientMain`, which runs the command line and all the communication over TCP;
- the **RMI thread**, which waits for the callbacks once the client's object has been exported and updates the local lists of users and chats;
- one **ChatSaver thread per chat**. When a callback brings a project that the user has just joined, the client starts a new saver for its chat; when a project disappears from the list, its saver is interrupted. Chats that are still present keep their saver and the messages it has collected.

When the server becomes unreachable, the client unexports its remote object, interrupts its savers and closes the socket before terminating, so that no thread is left behind.

7 Passwords

WORTH never stores a password. At registration, `HashPassword` derives a key from the password with PBKDF2 (Password-Based Key Derivation Function 2, [4]), using HMAC-SHA1 as the pseudorandom function, 20,000 iterations and a random salt of 32 bytes, and stores the salt and the 256-bit result, both in Base64, as a single string `salt$hash`. At login, the same function

is applied to the password received, with the stored salt, and the result is compared with the stored hash. Both ingredients have a purpose:

- the **salt**, different for every user, makes equal passwords produce different hashes, and makes precomputed tables of hashes useless;
- the **iterations** make every guess expensive: an attacker who obtained the file of the users would have to repeat the 20,000 iterations for each password tried.

After the session of Section 8, the entry of a user in `res/users.json` begins as follows (the two Base64 strings are 43 characters each):

```
{"nickname":"alice","password":"GIKjse7kt9gHC410cNWz...$di3lcoYypCwR8rwehQnA...","online":false, ...}
```

8 Verification

To verify the system end to end, the server was started on an empty folder and a scripted client drove it through the project's own protocol: it used the classes `Message`, `Request` and `Response` and the same JSON framing as `ClientCore`, registered through RMI, joined the multicast group of the project like a `ChatSaver`, and exported a `ClientInterface` object to receive the callbacks. The WORTH code was compiled and run unchanged. Table 4 reports the steps and the outcomes returned by the server.

Table 4: The scripted session: two users, one project, one card through its whole life cycle. Outcomes as returned by the server.

	User	Operation	Outcome
1–3	alice, bob	register alice, register bob, register bob again	OK, OK, USER_EXISTS
4–5	—	login with a wrong password; login of a user never registered	WRONG_PASSWORD, NOT_REGISTERED
6–7	alice	login; login again from a second client	OK, ALREADY_LOGGED
8–10	alice	create_project website, again, then list_projects	OK, PROJECT_EXISTS, OK
11–13	alice	add_member bob, bob again, an unregistered user	OK, MEMBER_EXISTS, NOT_REGISTERED
14–15	alice	add_card homepage, again	OK, CARD_EXISTS
16–17	alice	move todo → done, todo → toberevised	MOVE_CARD_FORBIDDEN (both)
18	alice	move todo → inprogress	OK
19–20	alice	move inprogress → todo, → backlog	MOVE_CARD_FORBIDDEN, NONEXISTENT_LIST
21–22	bob	login; show_members website	OK; [alice, bob]
23–25	bob	inprogress → toberevised → inprogress → done	OK (all three)
26	bob	move done → inprogress	MOVE_CARD_FORBIDDEN
27	bob	get_card_history homepage	TODO -> INPROGRESS -> TOBEREVISIED -> INPROGRESS -> DONE
28–29	bob	add_card tests; cancel_project website	OK, DELETE_FORBIDDEN
30–32	both	show_cards; logout alice and bob	homepage=DONE tests=TODO; OK, OK

Chat and callbacks. The project received the first address of the pool, 239.255.224.0. The scripted client, which joined its group after `list_projects`, received seven notices from the

service, one for each later event: the new member, the two new cards and the four moves of **homepage**, the last one announcing that bob moved it from **inprogress** to **done** (the notices are written in Italian in the source). Its callback object received four updates: the list of users, with alice online, when it registered for callbacks, and the list of projects three times, before and after the creation of **website** and after the addition of bob.

Persistence. After the session, **res** contained **users.json** and the folder **website** with **members.json**, **homepage.json** and **tests.json**. The server was then stopped and started again on the same folder, and a second script checked the restored state (Table 5).

Table 5: After a restart of the server on the same folder.

	User	Operation	Outcome
1-2	alice	login; list_projects	OK; [website], chat 239.255.224.0
3-4	alice	show_cards; get_card_history homepage	tests=TODO homepage=DONE; the full history
5-6	alice	move tests to inprogress, then to done	OK, OK
7-8	alice	cancel_project website; list_projects	OK; []
9	bob	login, then the socket is closed without logout	OK
10-11	bob	login from a new connection; logout	OK, OK
12	alice	logout	OK

Users, members, cards, lists and histories survived the restart, and the restored project received a chat address again. Once its last card reached **done**, the project could be canceled and its folder disappeared from **res**. The last steps confirm the handling of disconnections: when bob's connection was closed abruptly, the server logged him out, so his next login succeeded instead of answering **ALREADY_LOGGED**.

9 Building and running

The project was developed and tested with Java 8 and Java 15, on Unix and on Windows; for this report it was compiled and run, unchanged, with a current JDK. The three Jackson archives (version 2.9.7) are in the folder **lib** and must be on the class path. From the root of the project, on Unix:

```
javac -d out -cp lib/jackson-annotations-2.9.7.jar:lib/jackson-databind-2.9.7.jar:lib/jackson-core-2.9.7.jar src/*.java
java -cp out:lib/jackson-annotations-2.9.7.jar:lib/jackson-databind-2.9.7.jar:lib/jackson-core-2.9.7.jar ServerMain
java -cp out:lib/jackson-annotations-2.9.7.jar:lib/jackson-databind-2.9.7.jar:lib/jackson-core-2.9.7.jar ClientMain
```

On Windows the path separator is `;` instead of `:`. The server must be started first; any number of clients can then be started, each in its own terminal. A client started while the server is down terminates immediately with a message saying that the service is not available.

10 Conclusion

WORTH shows that a complete collaborative service can be built from the primitives of the Java standard library, provided that each one is used for what it does best. TCP gives every session a reliable channel for requests and responses, and a length prefix turns its stream of bytes into a sequence of messages. RMI makes registration a simple method call and, through callbacks, lets the server push changes to the clients as soon as they happen. IP multicast gives every project

a chat whose cost does not grow with the number of members, and a small pool of addresses makes it a resource the server can allocate and reclaim. NIO and JSON make the state durable and readable.

The concurrent side of the design is what makes these pieces work together: a cached pool that gives each client its own thread without slowing down the acceptance of new ones, synchronized blocks that make every check-then-update atomic, threads on the client that listen to the chats and the callbacks without blocking the user, and a careful handling of clients that disappear without notice. The scripted session confirmed all these behaviors on the running system, from the first registration to the restart of the server.

References

- [1] D. J. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010.
- [2] Oracle. *Java Remote Method Invocation Specification*. <https://docs.oracle.com/en/java/javase/21/docs/specs/rmi/>.
- [3] D. Meyer. *Administratively Scoped IP Multicast*. RFC 2365, IETF, 1998.
- [4] K. Moriarty, B. Kaliski and A. Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. RFC 8018, IETF, 2017.
- [5] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
- [6] FasterXML. *Jackson*, the Java JSON library. <https://github.com/FasterXML/jackson>.