Introduction

This experiment comprises DCM-RUST, a software architecture and supporting toolchain to facilitate runtime code mobility in Rust-based web applications. Code mobility, in simple words, refers to the ability to dynamically shift the execution location of a piece of code at runtime.

Overview

The overview of the DCM-RUST architecture is shown in the figure below:

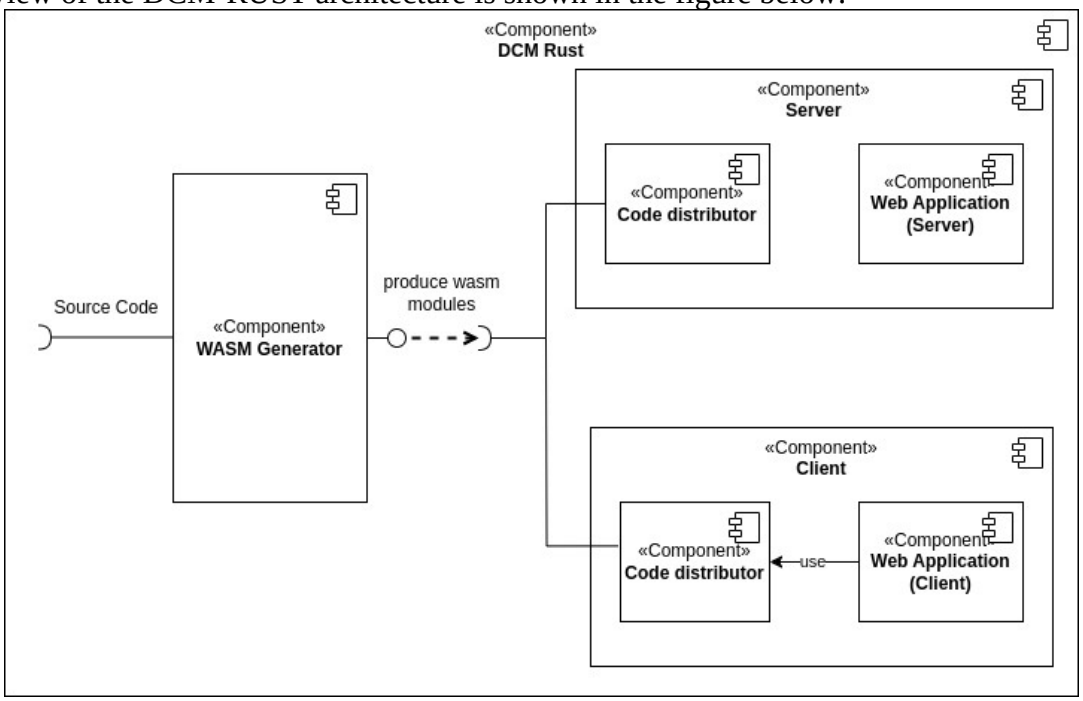

Figure 1: Overview of the DCM-RUST architecture

1. The developer specifies some portions of their application code (functions/objects) for mobility in their server-side code.
2. The WASM generator infrastructure automatically generates WebAssembly modules from the project's source code and moves them to the proper directories in client and server side code distributors.
3. Now, the developer can use these specified code fragments (functions/objects) on both the client and server side. Their execution happens on either the client or the server side depending on the current fragment distribution (their execution location).
4. For this experiment, the fragment distribution can be controlled through a chrome browser extension.

The demo application

In the experiment, you will be using a sample web-based shop to demonstrate the code mobility capabilities. It has basic features like browsing products, adding them to the cart, and placing orders. Figure below shows the main page of the demo application. Navigate to this link: http://dcm-rust-demo.cabwcegsewb3emeh.germanywestcentral.azurecontainer.io/ to see it live.

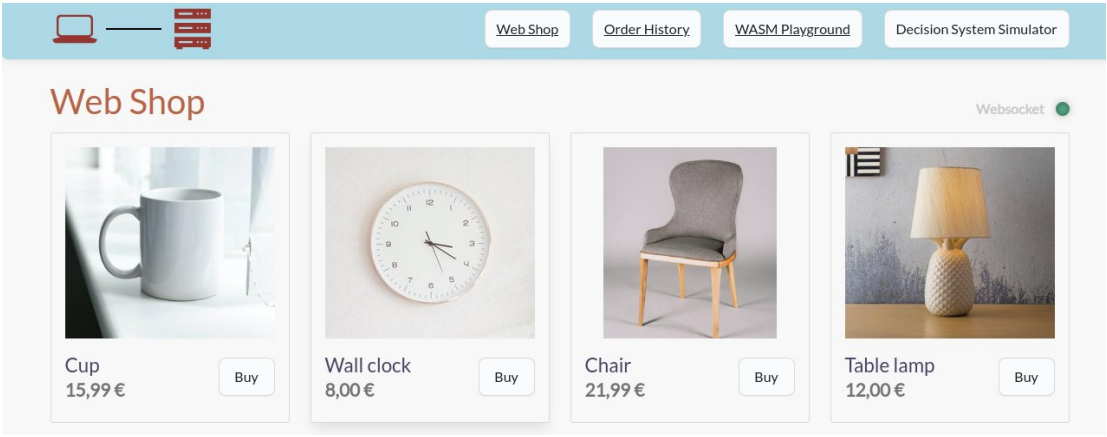

Figure 2: The demo application (Shopping page)

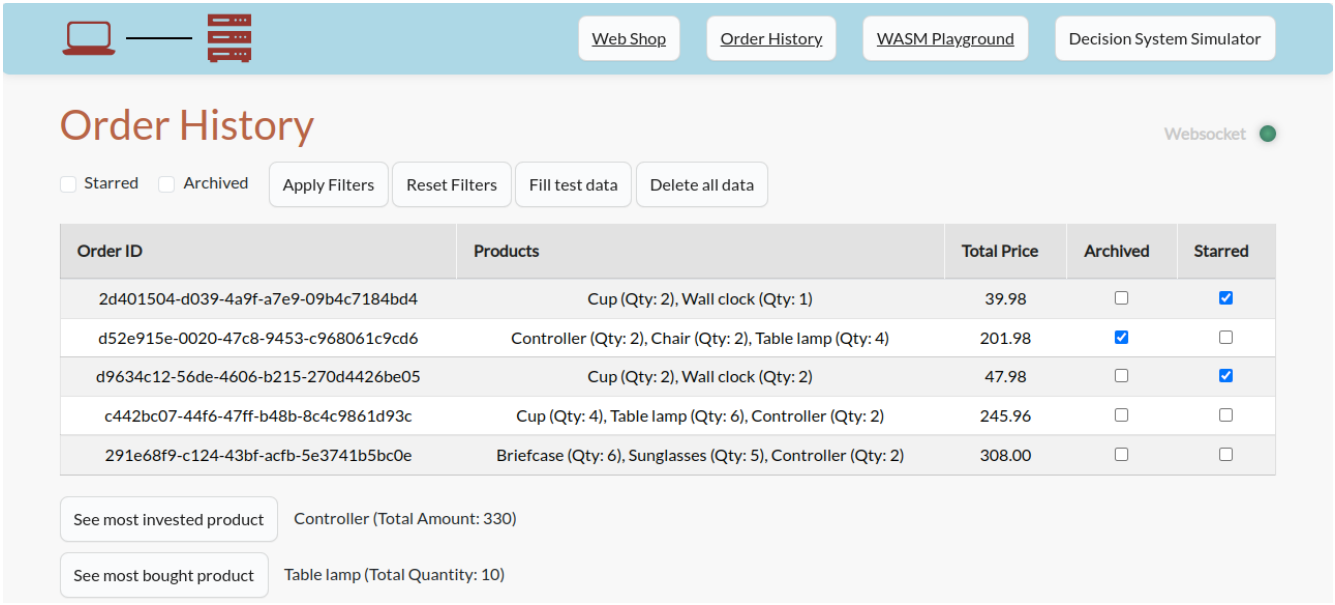

Figure 3: The demo application (Order history page)

After ordering some items, the user can then see their order history, as shown in figure above. Orders in the history can be starred or archived for later reference. The view can be filtered to show only the starred or archived orders. Below the order history table, there are two buttons labeled ‘See most invested product’ and ‘See most bought product’. When clicked, these buttons display the respective product along with relevant details like the total amount invested in that product or the total quantity bought just to its right. The most invested product is the one that the user has spent the most money on, and the most bought product is the one that is bought the most in terms of total quantity.

Task and Experiment

You are asked to perform some pre-defined tasks to get familiar with the system. You can access the codebase and required infrastructures in the cloud from a web browser with these details. The participant-id will be given to you by the researcher.

url: http://34.174.167.236:8080/?folder=/home/dx/project/dx/web-application{participant-id}

password: 26434b954bc74c77c11aa545

Shown in figure 3, the functions to find the ‘most invested product’ and ‘most bought product’ are currently implemented in JavaScript. You are asked to implement these functions in Rust, make them mobile and then replace their usage in JavaScript with the mobile functions. The actual logic of the functions is already provided in the form of helper functions in Rust. These helper functions take reference of the input as parameters, which prevents them from being made mobile (because references are not supported as parameters for mobile functions). So, your actual task is explained step-by-step below. There are three main phases, followed by a testing phase to conclude the experiment.

Step 1: Specification of mobile functions

In the first step, you will create two functions in the server-side code, and specify them for mobility.

1. Open the file: project_root/runtime-code-mobility-demo/src/shared/webshop /developer_experience_evaluation.rs.
2. At the bottom of this file, create two new functions. Name them **most_bought_product** and **most_invested_product** for convenience. The functions signature will be similar to **get_most_bought_product** and **get_most_invested_product** in the same file respectively. Only difference is that these new functions must take the actual value as parameter rather than references.
3. From these functions, call their respective helper functions by supplying the reference of the parameter (with the &), and return the value.
4. Mark the two newly created functions as mobile with ***/// @mobile*** doc comment on top of the functions.

This completes the first phase. You have created two functions and specified them for mobility using the ***/// @mobile*** doc comment.

Step 2: Generating WebAssembly modules

Now, you will run the wasm-generator infrastructure so that it automatically generates WebAssembly modules for the mobile functions.

5. Run the wasm-generator infrastructure from the web-ide’s terminal. Just type ‘wasm-generator’ and press enter. It takes roughly 3 minutes to complete.
6. When it finishes, the two functions from rust will now be available in the client side.

Step 3: Using the mobile functions

Finally, use the mobile functions (that you created in the server-side code in step 1) in the client-side code.

7. Open the file: project_root/runtime-code-mobility-demo/src/frontend/static/js/OrderHistory/index.js.
8. In this file, replace the function invocations for **get_most_bought_product** and **get_most_invested_product** with the relevant function you created by importing them from initCodeDistributor.js.

Step 4: Testing

To test if everything works as expected, follow these steps:

9. Start the code distributor and the demo application.
   - ➔ Open the web ide terminal in two tabs.
   - ➔ Navigate to project_root/code-distributor in one and to project_root/runtime-code-mobility-demo in another.
   - ➔ Execute command “cargo run” in both.
   - ➔ Wait for the project to compile and the web server to start in both terminals.
10. In your chrome browser, navigate to http://34.174.167.236:8081/.
11. A chrome extension is available as a zip file in the codebase. Download and install it in your browser. After installation select the extension and put the base url of code-distributor in the given input field, which is: http://34.174.167.236:8082/.
12. In the extension, click on fetch button. An ID is shown. Copy and paste it in the other tab in the extension. Click the ‘load’ button. Switch the location of one of the functions you created to ‘Server’, and click the ‘update’ button at the top.
13. Now, on the webshop application, in the Order History page, click on ‘Fill test data’ button to fill some test data. Afterwards, on the bottom of the page, click on ‘See most invested product’, and ‘See most bought product’. Verify that the server icon on the top of the page flashes for the one you just updated in the extension. It indicates that the execution took place on the server side. Also verify if the client icon flashes for the one you did not update. It indicates that the execution took place on the client side. Click them multiple times if you did not notice the flashing icon the first time.
14. Now, go to the extension again and directly to the second tab. Click the ‘load’ button. Switch the location of your previously updated fragment back to ‘Client’, and click the ‘update’ button at the top.
15. Finally, go back to the webshop application and refresh the page. Then, go to the Order History page, and click on ‘See most invested product’ and ‘See most bought product’ again. Verify that

the ‘Client’ icon flashes for both actions. Click them multiple times if you did not notice the flashing icon the first time.

Survey Questionnaire

That’s it for the experiment. Now, in your browser, head over to this link: https://bildungsportal.sachsen.de/umfragen/limesurvey/index.php/548967?lang=en, and fill the survey. Thank you for your participation!