DCM: Dynamic Client-Server Code Migration

Sebastian Heil[1][0000−0003−2761−9009] and Martin Gaedke[1][0000−0002−6729−2912]

Technische Universität Chemnitz, Chemnitz, Germany
{sebastian.heil,martin.gaedke}@informatik.tu-chemnitz.de

Abstract. The underlying Client/Server architecture of the Web inherently raises the question of the distribution of application logic between client and server. Currently, this distribution is static and fixed at design time, inhibiting dynamic and individual load distribution between clients and server at runtime. The benefits of dynamic migration allow balancing the needs of users, through increased responsiveness, and software providers, through better resource usage and cost reductions. Recent additions to the Web environment like WebAssembly provide a technological basis to move units of code at runtime. However, making use of them to extend a web application with dynamic code migration capabilities is challenging for web engineers. To that end, we devise a novel distributed Client/Server software architecture for web applications that supports dynamic migration of code at runtime, addressing the technical challenges of dependency management, distribution of control and data flow, and the required communication and interfaces. Our novel software architecture aims at providing a point of reference to web engineers seeking to extend their web applications with dynamic code migration capabilities and to contribute to the current re-consideration of the Web environment in the light of the standardization and wide-spread support of WebAssembly in all major browsers. Our experiments with 3 scenarios show that implementing such architecture is not only feasible but also that the impact on performance is negligible.

Keywords: Web Infrastructure · Software Architecture · Code Mobility · WebAssembly · WebSockets.

1 Introduction

Current Web applications are developed and executed on top of an established stack of technologies. The underlying Client/Server architecture of the Web inherently raises the question of the distribution of application logic between client and server. The design space for software architects is wide, ranging from relatively thin clients where most of the application logic is executed on the server side – e.g. making use of server-side MVC frameworks such as Django, ExpressJS, Laravel or Rails – to architectures in which more computations are run client-side in the browser and the server provides a minimal interface to the underlying data layer – e.g. with client-side frameworks such as React, Vue, Angular, or Svelte combined with high usage of AJAX.

While deciding the right distribution for a given web application depends on various factors and individual requirements, the distribution is static and fixed at design time. The mapping of units of code to either the client or server side is decided a priori and cannot be changed later dynamically at runtime, allowing to react to situational events and conditions. Especially in light of the ever-increasing heterogeneity of user devices on the client side, this static design time decision does not support balancing responsiveness/usability requirements by users, resource usage, and economic considerations by the software providers.

The availability of JavaScript on the server side via NodeJS and the support for executing server-side languages on the client side via WebAssembly [15], however, establishes more uniform client- and server-side platforms on top of which the vision of code mobility [1] at runtime becomes relevant and achievable for the Web. The benefits are a dynamic and individual distribution of load between clients and server as well as potential cost reductions for software providers.

The objective of this paper is to devise a novel distributed Client/Server software architecture for web applications that supports dynamically changing the location of execution of units of code at runtime. We address the technical challenges of dependency management and compilation, distribution of control and data flow, and the required communication and interfaces and propose solutions for each of the challenges. The resulting architecture as well as a supporting infrastructure was implemented and put to test in several scenarios.

The remainder of this paper is structured as follows: in section 2 we outline our proposed solution architecture and the supporting infrastructure, section 3 positions our work against existing code mobility paradigms and approaches, in section 4 we evaluate the feasibility of the architecture in 3 scenarios and show that the performance overhead is negligible, and section 5 concludes the paper with an outlook on directions for future work.

2 The DCM Architecture

In this section, we present our solution to enable dynamic code migration between client and server at runtime based on standardized Web technologies. We propose a novel software architecture – the DCM Architecture – that empowers Web Engineers to add dynamic code migration capabilities to the web applications they build with a dedicated focus on minimizing the impact/requirements on the development activities. Figure 1 provides an overview of the main components of the DCM Architecture and their interactions with each other. The DCM Architecture specifies a supporting dynamic migration infrastructure (in blue) that can be embedded into a web application (in black), on top of which Web Engineers can control the migration through simple configuration. Our approach comprises solutions to three main challenges:

1. the specification and compilation of migratable code fragments for client and server,
2. the orchestration of execution/control flow of these fragments between client and server, and the

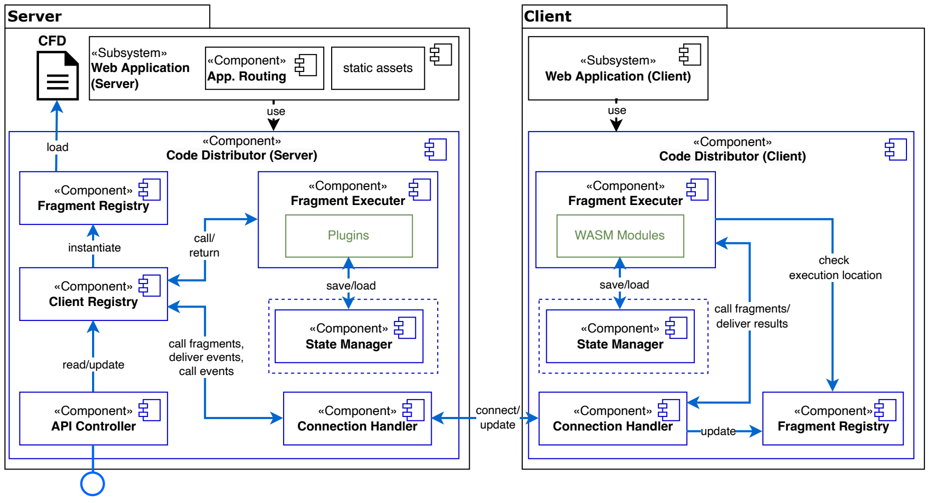
Server
CFD
«Subsystem» Web Application (Server)
«Component» App. Routing
static assets
use
load
«Component» Code Distributor (Server)
«Component» Fragment Registry
«Component» Fragment Executer
Plugins
call/ return
instantiate
«Component» Client Registry
save/load
«Component» State Manager
read/update
call fragments, deliver events, call events
«Component» API Controller
«Component» Connection Handler
connect/ update
Client
«Subsystem» Web Application (Client)
use
«Component» Code Distributor (Client)
«Component» Fragment Executer
WASM Modules
check execution location
save/load
call fragments/ deliver results
«Component» State Manager
«Component» Connection Handler
update
«Component» Fragment Registry

Fig. 1. Main Components and Interactions of the DCM Architecture enabling Dynamic Code Migration between Client and Server at Runtime. Supporting Infrastructure is Highlighted in Blue, Automatically Generated Artifacts are Highlighted in Green.

3. synchronization of fragment distribution information and redirection of data flow at runtime.

The following subsections detail our solutions to these three challenges.

2.1 Generation and Compilation of Code Fragments

In this subsection, we outline our concept of migratable code fragments, how Web Engineers can specify these parts of the codebase to be executable on server and client side, the required metadata and its semi-automatic extraction, as well as the validation, compilation and deployment of executable modules from the specified code fragments.

Specification of Code Fragments. To allow the Web Engineer to specify migratable subsets of the web application's codebase that can be dynamically moved between client and server at runtime, we define these as *Code Fragments*. A code fragment $CF = (D_i, L, T, M)$ consists of the specification of its source document $D_i \in \mathfrak{C}$ within the codebase $\mathfrak{C}$ and limits $L = (\alpha, \omega)$ (line numbers $\alpha, \omega \in \mathbb{N}$) within D_i, its type $T \in \{function, variable, typedefinition\}$ and the migration-relevant metadata M. Limits can be expressed either via code annotations in the source code itself or externally by numerical specification. In DCM, the level of granularity for specifying executable code fragments is on the level of individual functions to provide a balance between fine-grained control (i.e. smaller re-use units than components/classes) and isolation/dependency management (i.e. larger than sets of statements). Fragments for variables and

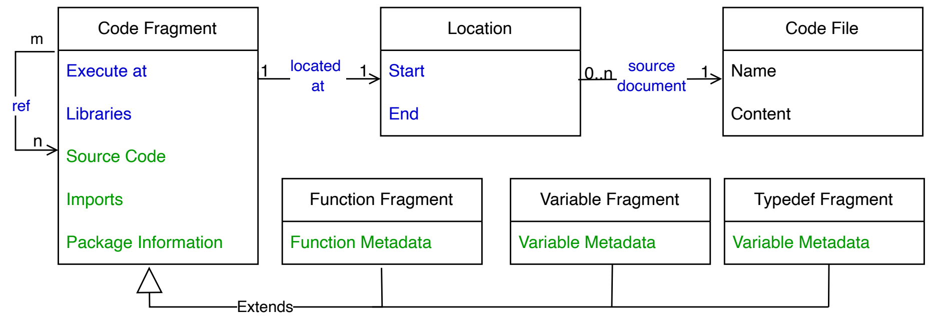

Fig. 2. Code Fragments Data Model. Web Engineer provided information in blue, automatically identified information in green.

type definitions are required to handle imports of the functions but are not executable on their own.

As shown in fig. 2 a fragment aggregates Web Engineer-provided information and information automatically identified through static code analysis of the codebase. In addition to location and fragment type information D_i, L, T, Web Engineers specify the intended initial execution location (server or client), libraries used, and referenced other fragments. The automatically identified information comprises the actual source code as specified by the location information, imports from other sources, package information, and structural information of functions/variables/type definitions such as function parameters. Information about all code fragments is aggregated in the artifact denoted as CFD (code fragment description) in fig. 1. A sample excerpt of the manually specified parts of a CFD can be seen in listing 1.1.

Syntax Analysis allows to automatically derive information from the codebase necessary for the compilation of code fragments without requiring manual specification by the Web Engineer and thus significantly reducing the required effort. Our approach operates on the *abstract syntax tree (AST)* resulting from parsing the source code and identifying tokens and their relations according to the grammar of the programming language in use. Generating the AST from the source code requires language-dependent tooling. To extract the data about imports from other sources, package information, and structure of functions/variables/type definitions, the AST is traversed, and the collected data merged with the specifications manually made by the Web Engineer.

```
fragments:
...
  - id: 6
    name: GetHash
    runOn: client
    location: { filepath: shared/shared.go }
    libs: [crypto/sha256, encoding/hex]
```

```
      ...
  - id: 21
    name: CreateEmployee
    runOn: client
    dependsOn: [6]
    location: { filepath: shared/employee.go }
...
```

Listing 1.1. Sample excerpt of a Code Fragment Description

Compilation of Code Fragments. To support execution of code fragments specified by the Web Engineer in the CFD on server and client, they need to be compiled for both target platforms. The compilation target on the server side depends on the specific language and platform (typically shared objects, dynamic-link libraries etc.), the compilation target on the client side is WebAssembly. Server-side compiled artifacts are managed as plugins, client-side compiled artifacts as WebAssembly modules, both of which can be dynamically loaded at runtime.

The compiler input for building the language-specific plugins and WebAssembly modules is automatically created based on the codebase and the metadata of each code fragment. As parts of this information are user-provided, a `CFD Validator` checks the structural validity and completeness of the CFD. Errors such as duplicate fragments or missing required information are reported with additional debug information so that the Web Engineer is supported in fixing them. If validation is passed, automatic code generation and transformation is performed. The generated code for plugin and WebAssembly fragments handles imports and dependencies to turn them into separately compilable units of code. This requires consideration of all fragments' metadata together as duplicate imports resulting from dependency chains need to be resolved. The existing codebase is modified so that invocations of code declared as migratable fragment are redirected to the `Fragment Executer`. This allows executing either the server-side plugin or forwarding the control and data flow to the client-side DCM infrastructure to be handled by the corresponding WebAssembly module. DCM infrastructure components (c.f. `Code Distributor` in fig. 1) are injected. The resulting modified codebase is then compiled for server and client and plugins and WebAssembly modules (including JS glue code) are moved to the correct directories for availability for the server and client side code distributor. In particular, WebAssembly modules need to be deployed to the directory serving static assets (JavaScript, CSS) so they can be loaded via HTTP(S) on the client side.

2.2 Dynamic Migration of Code Fragment Execution

To enable the execution location of code fragments to dynamically change at runtime, the DCM architecture specifies infrastructure that handles their life cycle, and the distribution of control and data flow. These are the `Code Distributor` components on the server and client side (cf. fig. 1). They manage the loading, execution, and termination of plugins/WebAssembly modules respectively, and synchronize the required incoming/outgoing data flows and events.

Code Distributor (Server). Architecturally, the `Code Distributor` component can be either embedded with the web application itself or an external stand-alone proxy-like server process, potentially on a different host. Unlike approaches like HTML5 Agents [14], we propose a direct embedding due to lower required resources/operations and maintenance efforts and lower communication complexity. Embedding requires adding the DCM library to the web application's imports and registering routes for the `Code Distributor` in the application-internal routing. These steps can be automated in the codebase modifications of the compilation step described in section 2.1. The `Code Distributor` checks incoming client requests and signals connection errors. For valid requests, it performs session management using a `Client Registry`. Together with the `Fragment Registry`, it keeps track of all code fragments and enables different individual fragment distribution patterns per client. Both registries are initialized from the Web Engineer's specifications of available code fragments and their initial execution location in the CFD. The fragment distribution status is synchronized with each client's `Code Distributor`, updates in the fragments' execution location trigger the required steps for control and data flow migration via the `Fragment Executer`. To execute a fragment on the server side, the `Fragment Executer` loads and executes the compiled plugin fragment. The fragment's state is monitored by the `State Manager` to be able to restore it after migration. In particular, this comprises changed/initialized variables, loops, and time functions. Changes in resources shared with other fragments are synchronized. The `API Controller` provides a RESTful interface to monitor and control the dynamic code migration. It interfaces the fragment distributions per client and allows to change them in order to trigger a migration at runtime. This enables scenarios in which an automated decision system can determine optimal distributions based on runtime measurements of load, network bandwidth, etc.

Code Distributor (Client). The client-side DCM infrastructure mirrors the `Fragment Executer` and `State Manager` components described above for execution and state management. While the server-side `Code Distributor` is implemented in the web application's backend language, its client-side equivalent is implemented in JavaScript and served via the application's static web resources. The `Fragment Executer` handles loading, initialization, and invocation of fragments previously compiled to WebAssembly modules and the data conversions between JavaScript and backend language's type system within the WebAssembly modules. Information about the execution is retrieved from the `Fragment Registry` before each invocation, which is synchronized with the server side. Similar to offloading approaches like MAUI [4] and ThinkAir [9], fragments are initially executed locally until updates are received to enable execution during connection establishment and initialization. Likewise, connection losses lead to local client-side execution as fallback. Fragments are executed as tasks via a pool of WebWorkers.

2.3 Client/Server Synchronization and Data Flow Redirection

The DCM code distributor components need to constantly exchange information bidirectionally in order to synchronize fragments and distribution state and to maintain the data flow from and to fragments when they change their execution location. This subsection outlines the corresponding communication channels and protocol. We propose to employ WebSockets for communication, offering a bidirectional connection between client and server-side DCM infrastructure, as it currently is supported by more browsers than the new WebTransport W3C standard. Messages within the WebSocket connections are represented in JSON. The `Connection Handler` components establish client-server WebSocket connections on initialization. Clients identify themselves via JSON Web Tokens (JWT) which are included in all communications to allow the server to handle fragment distributions for each client individually. Client and server communicate by exchanging events represented as JSON via the WebSocket connection.

Table 1 shows the main events of the DCM communication protocol. There are events for two different purposes: a) to exchange information required for the management of fragments and distribution state, and b) to enable the redirection of data flow for fragments where caller and callee are not on the same side.

Through the codebase modifications described in section 2.1, both client and server `Fragment Executer` serve as proxies between the caller of a fragment and the fragment code itself. Data flows in and out of the fragments via function parameters, return values, and global variables. The information about these is gathered and contained in the CFD as described above. At runtime, each `Fragment Executer` checks the current execution location of the called fragment and redirects the data flow if the location is not local. The `callFunction` event allows specifying an optional `defer` property that tells the `Fragment Executer` to execute the invoked fragment in the background, enabling other fragments to be executed during a long-running computation in the initially called fragment.

Table 1. DCM Communication Protocol Events.

Name	Payload	Description
updateFragments	[object] fragmentStatus-List	triggered whenever there is a change in the fragments' distribution to synchronize Client and Server Distributor
callFunction	string funcName, [object] params, boolean defer?	triggered when a fragment is called remotely, to pass incoming data to the called fragment
functionResult	string funcName, object result	triggered when a fragment invokation yields a result to return it to the caller

3 Related Work

Code mobility has long been a topic of research interest for distributed systems [1]. While, *Code on Demand* is the pre-dominant code mobility paradigm on the Web, the availability of NodeJS and the *WebAssembly* standard [15] has opened opportunities for other paradigms such as *Remote Evaluation* and *Mobile Agents* in the recent Web environment.

Code on Demand [1] is the most widely used code mobility paradigm in Web applications. The ability of the client to request and execute code from the server at runtime is supported via HTML script tags allowing to load JavaScript files and execute them in the browser. Popular client-side frameworks like React, Vue, Angular, or Svelte imply an architecture in which JavaScript code is loaded at runtime and dedicated infrastructure – Content Delivery Networks (CDNs)– to support faster provision of the most commonly loaded framework code artifacts are used. Most prominently, the Code on Demand mobility paradigm was included as the sixth architectural constraint of REST [6] which forms the architectural blueprint for many current Web applications. Beyond common Code on Demand practice, Sparkle [13] additionally supports capturing, migrating, and restoration of application state. Unlike the DCM architecture, Code on Demand mobility is unidirectional, from server to client, and, while the actual code artifacts are migrated at runtime, the decision to execute them on the client side is fixed at design time for common Web applications using JavaScript, or, for recent platform-dependent approaches like Blazor[1], Web Assembly.

Remote Evaluation [1] approaches allow a unidirectional change of the execution location of a unit of code from client to server. Offloading approaches like MAUI [4], CloneCloud [3], or ThinkAir [9] focus on supporting computationally weaker mobile devices by running computations on the server. Like DCM, MOJA [2] and PIOS [12] make use of a uniform platform of client and server side and MOJA also uses WebSockets for communication. Unlike the DCM architecture, however, the direction is only from client to server, and due to building on NodeJS, they are bound to JavaScript, while the use of WebAssembly modules in DCM potentially enables using and migrating arbitrary languages. Both Code on Demand and Remote Evaluation approaches do not consider redirections of data flow or transfer of state as they focus on unidirectional code mobility.

Mobile Agents [1] approaches are focused on moving entire software components at runtime across the network. Telescript [5] was an early approach that enabled mobile agents by providing a dedicated object-oriented language supporting the migration of objects as software agents to other *places* at runtime, including the ability to interrupt the execution of an agent and continue it in the new place. Instead of requiring a dedicated language, many mobile agents approaches like Java Aglets [10] employ the threading and networking capabilities of the Java platform [1]. Aglets combine Java Aplets and Servlets, specifying lifecycle methods for Java objects that support their creation, cloning, dispatching, retraction, activation/deactivation and messaging to move them between

[1] https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

client and server. Execution of Aglets can be paused and resumed with the previous state in a new location. HTML5 Agents [14] implement the mobile agents paradigm on top of standardized Web technologies: HTML5, CSS, JavaScript. It makes use of the platform uniformity on client and server through NodeJS. Liquid.js [7] is another contemporary mobile agents framework making use of Web standards such as WebRTC and WebWorkers. It focuses on a seamless user experience for moving Web Components, via polymer.js, across multiple heterogenous devices to allow them to "follow" the user. Disclosure [8] addresses the JavaScript-runtime-specific challenge in cross-device liquid computing to handle closures. To enable execution state of the migrated component, it proposes an instrumentation-based technique that has a limited runtime penalty of 0-15%. In comparison to DCM, mobile agents approaches have a lower level of granularity, moving components as a whole, including its code. The moved component therefore no longer exists at its origin and moves freely between peer hosts. The implied design-level security challenges of mobile agents [1] are less severe in DCM, as code migrates only between client/server of the same application and its redundancy means that "new code" cannot be easily introduced. The lack of an established platform particularly in browsers is considered an important reason for the limited success of mobile agents [1]. DCM leverages the changed situation of wide browser support for WebAssembly similar to containers in [11].

4 Evaluation

To evaluate the proposed DCM architecture and infrastructure, we instantiated these using the language platform and compiler toolchain of Go. Our experiments are based on three scenarios that put different aspects of the architecture and infrastructure under test to investigate the performance impact of the proposed solution. In the following, we describe the material and evaluation procedure, report the obtained results, and discuss them.

4.1 Material and Procedure

The Go-based evaluation implementation of DCM is available online for review. To facilitate and homogenize repeated evaluation runs and measurements under equal conditions, the evaluation setup was container-virtualized using Docker. Our experimental evaluation comprises three scenarios with different configurations each. Scenario I covers the analysis and compilation steps necessary to create executable modules described in section 2.1. Scenario II tests the required network communication between the client and server side `Code Distributor` components with regard to stability and delays. Scenario III evaluates the behavior of migration and fragment execution at runtime. All evaluation materials and test scripts are provided online[2]. The test runs of the scenarios used the following hardware: Server: AMD EPYC Processor (2 CPUs @ 3.7Ghz), 4GB RAM,

[2] https://github.com/heseba/dcm

Debian Linux 10 64-bit. Client: Intel Core i5-4690K (4 CPUs @3.5GHz), 16GB RAM, Windows 10 Pro 64-bit (Build 19044). Network delay between server and client measured via ping ranges between 16.5ms and 44.6ms.

Scenario I. This scenario investigates the analysis and compilation of code fragments and the impact of different codebase sizes on each involved component. Two popular open source projects in Go language were selected as material from Github: wire[3] at about 200 and terraform[4] at about 1200 Go source files. For terraform, a docker configuration was created to simulate the settings for integration into a project by a web engineer. Additionally, a custom test suite in Go with 7 Go files and a high number of fragments dedicatedly testing different computations, errors, and data type handling was created. For all three sample projects, CFDs were manually created (wire: 5, terraform: 6, custom: 37 fragments). For each of the three samples, the test script would perform the three automatic steps of *code analysis*, *fragment generation*, and *compilation*. Each step was repeated five times and execution time was captured.

Scenario II. This evaluation scenario tests the network connections and communication timings between server and client side of the DCM architecture. To that end, the timings for different cases are measured: the round-trip time for an echo signal between client and server, the time until the initial WebSocket connection to the client is established and it has received the first fragment list update and the time between receiving a migration command via the server-side interface and the client receiving the new information. The measurements are implemented in a custom test application using a modified version of the DCM infrastructure with extended access to the WebSocket connection and inbuilt time measurements. *Echo time* is measured by the client sending an echo event to the server until receiving the response. *Initialization time* measurement is triggered by a page reload on the client, requesting the list of fragments and execution locations until it receives it. *Command time* is measured from the server side receiving a location update command via the API, forwarding the information to the client until it is received.

Scenario III. This scenario showcases the migration and fragment execution at runtime. It tests the execution results before and after execution of fragments on server and client. To simulate long-running, side-effect-free computations, two algorithms were implemented in Go and JavaScript: Fibonacci and nth prime. These are run as server-side fragment plugin and as client-side WebAssembly module, as well as in plain JavaScript as baseline. The scenario comprises three test cases: *Single* tests the computation of the Fibonacci to $n = 100$. *Iterated* tests the computation of the Fibonacci to $n = 93$ (the limit of Go's `int64`) repeated in a loop for 1000 times. *Prime* tests the behavior for optimized vs. non-optimized execution conditions by computing the 500.000th prime number. The optimized conditions run 50 iterations within the fragment itself, whereas non-optimized execution invokes the fragment 50 times. During the longer executions, the system behavior by incoming migration commands was observed.

[3] https://github.com/google/wire
[4] https://github.com/hashicorp/terraform

4.2 Results

Scenario I. Table 2 shows the time measurements for scenario I for the three samples per each step collected in 5 test runs $r_{I,1}$ to $r_{I,5}$. The test runner script was executed in a running docker container. All test runs were completed successfully. The DCM infrastructure assisted the CFD specification by showing hints, e.g. for missing dependencies or attributes. All fragments could be successfully compiled and were automatically deployed to the correct directories.

Table 2: Scenario I Measurements: test runs $r_{I,1}$ to $r_{I,5}$

Sample	**Step**	$\boldsymbol{r_{I,1}}$	$\boldsymbol{r_{I,2}}$	$\boldsymbol{r_{I,3}}$	$\boldsymbol{r_{I,4}}$	$\boldsymbol{r_{I,5}}$
Wire	code analysis	41ms	42ms	39ms	39ms	37ms
	frag. generation	48ms	48ms	53ms	49ms	55ms
	compilation	18ms	18ms	19ms	19ms	21ms
Terraform	code analysis	1,841ms	1,589ms	1,559ms	1,577ms	1,542ms
	frag. generation	192ms	178ms	179ms	176ms	174ms
	compilation	457ms	436ms	421ms	410ms	386ms
Custom	code analysis	3ms	3ms	3ms	3ms	3ms
	frag. generation	63ms	59ms	59ms	61ms	61ms
	compilation	13,734ms	7,198ms	7,131ms	7,413ms	7,141ms

Scenario II. Table 3 shows the time measurements for the network communications of scenario II for each test case collected in 5 test runs $r_{II,1}$ to $r_{II,5}$. Additionally, we tested connection recovery behavior. The mean recovery time from a connection loss was 15s. After a client-side page reload, the system could successfully reconnect the `Code Distributor` components of client and server in presence of several other connected clients. The mean roundtrip time was 21ms ($\sigma = 1ms$). Initialization took 42.8ms on average ($\sigma = .45ms$). A fragment update command was distributed in 43.8ms ($\sigma = .45ms$).

Table 3: Scenario II Measurements: test runs $r_{II,1}$ to $r_{II,5}$

Time Measurement	$\boldsymbol{r_{II,1}}$	$\boldsymbol{r_{II,2}}$	$\boldsymbol{r_{II,3}}$	$\boldsymbol{r_{II,4}}$	$\boldsymbol{r_{II,5}}$
Echo time	22ms	20ms	22ms	20ms	21ms
Initialization time	43ms	43ms	43ms	42ms	43ms
Command time	44ms	43ms	44ms	44ms	44ms

Scenario III. Table 4 shows the time measurements of the three test cases in different configurations. All measurements are with browser-side caching enabled and excluding the initial loading times of WebAssembly modules. These were measured at a mean of 44ms ($\sigma = 12ms$). Fragments could be successfully migrated between server and client. Computed fragment execution results of WebAssembly modules and server plugins were always identical.

Table 4: Scenario III Measurements: number of elements and iterations and overall execution times for JavaScript, WebAssembly and Server plugins

Case	**Elem.**	**Iter.**	**JS**	**WASM**	**Plugin**
Single	100	1	0.1ms	60.6ms	28.1ms
Iterated	93	10	0.3ms	638ms	246ms
	93	1,000	30ms	69,326ms	24,179ms
Prime	100,000	10	2.21s	3.8s	0.74s
	500,000	50	132.56s	185.04s/ 159.45s*	21.03s

* optimized

4.3 Discussion

Scenario I. The measurements show that analysis and compilation times depend on the complexity of the codebase. Expectedly, the more files in the codebase, the longer the AST-based analysis time and the time to generate the fragment code due to the higher number of comparisons between CFD and the fragment code. However, even for larger projects like Terraform with about 1200 code files analysis times remain well under 2 seconds and generation times well below 1 second. The compilation time directly depends on the number of fragments from the generation step. The impact of the overall times through our infrastructure on the regular compilation and deployment activities for Web applications can be considered negligible, especially as the added time does not occur at runtime but at design time and thus less frequent. In our experiments, we further observed some difficulties to manage the fragment IDs manually. This could be improved through a UI with automatic ID assignment or non-integer expressive fragment names as identifiers. No further difficulties were encountered in the three test cases and the DCM infrastructure's hints with automatic correction proposals facilitated executing scenario I.

Scenario II. Our experiments show that the underlying WebSocket connections could be reliably established and managed for different clients. Automatic re-connects improve the user experience in presence of network errors. The relatively low and constant times for all three test cases, placed well within the range of the test setup's raw network delay (16.5-44.6ms), exhibit a very low impact on the runtime performance of a Web application implementing the DCM architecture and are barely noticeable by end users.

Scenario III. WebAssembly plugins on the client side exhibit higher execution times compared to native JavaScript and server plugins for all three test cases. This is particularly due to the additional loading time of WebAssembly modules, even with caching enabled. Considering the WebAssembly execution time without the mean loading time of 44ms, results are much closer to the server plugin times. Apart from not requiring comparable initial loading times, the advantage of native JavaScript over both fragment types for lower numbers of elements and iterations stems from the speculative optimizations of V8 runtime's JIT compiler Turbofan, creating shapes for monomorphic functions. With higher numbers of elements and iterations, this advantage reduces relative to the advantage of compiled language execution compared to interpretation, leading to better times for server plugins and similar times with WebAssembly (excluding module loading times as mentioned above).

Threats to Validity. Our experiments aim at providing a proof of concept for the feasibility of the DCM architecture and some first insights on its implementation supported by the DCM infrastructure. *Internal validity* can be threatened by the execution of the experiments' manual activities by one single student assistant familiar with the architecture and toolchain. However, we are not making particular claims about Web Engineers' experience, effort, or difficulty, which would require a user study involving developer test subjects with diverse demographics. The specific choice of public real-world projects in scenario I can have influenced the results, but we selected two popular Go projects from Github at different sizes and from different domains. All reported time measurements were automatically measured based on integration in the scenarios' code without the potential for subjective biases. Furthermore, all evaluation materials are available on GitHub for replication. *External validity* of our experiments is limited specifically by the choice of the Go ecosystem. While valid for demonstrating feasibility, the measurement results cannot be generalized to other WebAssembly-compilable languages, as the code analysis, fragment generation, and compilation are dependent on the available AST parsers, compilers, and language features. For that reason, further experimentation with other languages is required for a more general understanding of the DCM architecture. Generalization of our results beyond feasibility, e.g. concerning the applicability of DCM in different application domains, is not intended and would require dedicated experimentation with qualitative empirical approaches. *Construct validity* is threatened for the command time measurements of scenario II. Unlike measurements of scenario I, III, and for echo time and initialization time, the command time is based on start and end time stamps from two different host systems. While a dedicated study of performance would require more sophisticated instrumentation to synchronize the system clocks, we argue that the command time measurements are still valid to show the general dimension of the impact of DCM. We do not make specific numerical claims for these measurements beyond. Both hosts' clocks were synchronized via NTP with an expected prevision in the 5-100ms[5] range, so that even for large synchronization differences command times would reach a maximum of 250ms, not violating our claims to low impact on performance perceived by end users.

[5] cf. http://www.ntp.org/ntpfaq/NTP-s-algo.htm

5 Conclusion & Future Work

In this paper, we proposed a novel software architecture for Web applications enabling dynamic migration of code fragments at runtime. It enables differing fragment distributions individually per each client-server pair to better adapt to situational availability of client/server resources. Unlike previous code mobility approaches, it is platform-agnostic, leveraging W3C standardized and established technologies such as WebAssembly and Web Sockets. Based on extensive experimentation with the DCM architecture, we provided insights on how to address the main challenges of specification and compilation of migratable code fragments, orchestration of fragment execution control flow, and distribution management and redirection of data flow at runtime. To that end, we devise a supporting infrastructure that supports Web Engineers to make use of DCM. To evaluate the proposed architecture and infrastructure, we reported and discussed the results of experimentation with three different scenarios addressing different architectural aspects and performance impact of the solution. All implementation and experimental materials are provided to allow the Web Engineering community to replicate our experiments and extend the approach. Our experiments provided a proof of concept for the feasibility of the DCM architecture, some first insights on its implementation supported by the proposed infrastructure, and indicate that the impact on performance from the required fragment management and communication can be limited to negligible levels.

While these results are promising and showcase the potential of the WebAssembly standard for extending the capabilities of the current Web application infrastructure in the context of code mobility, we also identified some limitations and directions for future work. Running well for migrating atomic, side-effect free functions at runtime, state management, a code mobility challenge for a long time[1], is hard the more stateful these functions are. Suitable mechanisms for interruptions of long-running functions with restoring of internal states are an open challenge the achievability of which depends also on the specific language platforms. Also, due to the current limitations of exchanging data between JavaScript and WebAssembly as numerical data via the Heap, fragments with data flows comprising complex structured data that cannot be easily transformed, require the development of more sophisticated serialization techniques or potential future extensions of the WebAssembly standard for supporting object transformations. Conceptually our work establishes a basis for the dynamic migration of code fragments at runtime. To fully leverage the benefits of situationally balancing responsiveness/usability requirements by users with resource usage and economic considerations by software providers for each individual client, an automatic decision system is required. It could optimize the fragment distribution at runtime based on information on the individual hardware capabilities and, particularly, through measurements of current load on client and server side, interacting with DCM via the provided API. The creation of such a decision system and the integration with dynamic code migration like in DCM are promising directions for future research to which we plan to contribute first insights from currently ongoing experiments.

Acknowledgements The authors would like to thank Alexander Senger for his valuable contributions to the proof-of-concept implementation of the DCM architecture and the evaluation experiments.

References

1. Carzaniga, A., Picco, G.P., Vigna, G.: Is Code Still Moving Around? Looking Back at a Decade of Code Mobility. In: Proc. of ICSE'07 Companion. pp. 9–20. IEEE (2007)
2. Chaoran Xu, Murray, N., Yuansong Qiao, Lee, B.: MOJA - Mobile Offloading for JavaScript Applications. In: Proc. of ISSC/CIICT 2014. pp. 59–63. Institution of Engineering and Technology (2014)
3. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: CloneCloud: Elastic Execution between Mobile Device and Cloud. In: Proc. of 6th EuroSys. p. 301. ACM Press, New York, New York, USA (2011)
4. Cuervo, E., Balasubramanian, A., Cho, D.K., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: MAUI: Making Smartphones Last Longer with Code Offload. In: Proc. of 8th MobiSys. p. 49. ACM Press, New York, New York, USA (2010)
5. Domel, P.: Mobile Telescript agents and the web. In: COMPCON '96. Technologies for the Information Superhighway Digest of Papers. pp. 52–57. IEEE Comput. Soc. Press (1996)
6. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine (2000)
7. Gallidabino, A., Pautasso, C.: The LiquidWebWorker API for Horizontal Offloading of Stateless Computations. Journal of Web Engineering **17**(6), 405–448 (nov 2019)
8. Kim, J.Y., Moon, S.M.: Disclosure: Efficient Instrumentation-Based Web App Migration for Liquid Computing. In: Web Engineering. Proc. of ICWE 2022. pp. 132–147. Springer, Cham (2022)
9. Kosta, S., Aucinas, A., Pan Hui, Mortier, R., Xinwen Zhang: ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In: Proc. of IEEE INFOCOM. pp. 945–953. IEEE (2012)
10. Lange, D., Oshima, M.: Mobile agents with Java: the Aglet API. World Wide Web **1**, 1–18 (1998)
11. Mäkitalo, N., Mikkonen, T., Pautasso, C., Bankowski, V., Daubaris, P., Mikkola, R., Beletski, O.: WebAssembly Modules as Lightweight Containers for Liquid IoT Applications. In: Proc. of ICWE2021. pp. 328–336. Springer, Cham (2021)
12. Park, S., Chen, Q., Yeom, H.Y.: PIOS: A platform-independent offloading system for a mobile web environment. 2013 IEEE 10th CCNC pp. 137–142 (2013)
13. Siu, P.P., Belaramani, N., Wang, C.L., Lau, F.C.: Context-aware state management for ubiquitous applications. In: Embedded and Ubiquitous Computing. EUC. vol. 3207, pp. 776–785. Springer Verlag (2004)
14. Voutilainen, J.p., Mattila, A.l., Systä, K., Mikkonen, T.: HTML5-based mobile agents for Web-of-Things. Informatica **40**(1), 43–51 (2016)
15. WebAssembly Community Group: WebAssembly Specification — WebAssembly 2.0 Draft 2022-12-15 (2022), https://webassembly.github.io/spec/core/