
XM_0128 - FUNDAMENTALS OF ADAPTIVE SOFTWARE









Name of the Study:



EMERGENT WEB SERVER



TEAM: GREEN LIGHT DISTRICT

Student Name	Student Number
	
	
	
	

GitHub Repositories

EWS: 

UPISAS: 

TABLE OF CONTENTS

SECTION	TITLE	PAGE
1	INTRODUCTION	3
2	REQUIREMENTS	3
3	INSTALLATION AND SETUP	4
4	IMPLEMENTATION	7
5	TESTING	11
6	DIVISION OF WORK	13

SECTION 1 - INTRODUCTION

The Emergent Web Server (EWS) is a dynamic web server capable of serving HTTP 1.0 requests through its 42 unique runtime configurations. It is engineered to adjust to changing demand by reassembling its components on the go, ensuring a seamless transition between configurations without missing or dropping requests. This capability allows it to maintain robust service continuity even as it adapts to varying workload patterns.

EWS offers a variety of stream processors, which include different functionalities such as compression and caching, using several algorithms to optimize performance. These processors are part of what allows EWS to fine-tune its configurations according to the needs of the moment.

An integral part of EWS is its RESTful API, which grants external systems the power to inspect the web server's current state, enact changes to its configuration, and gather important performance metrics, all in real-time. This API is a critical tool for managing the web server's operation and ensuring optimal performance.

To aid in the use and research of EWS, the PyEWS Python module has been developed. This module simplifies the process of interacting with EWS for developers and researchers. With the included baseline online learning algorithm, users of EWS can evaluate different learning strategies to determine their effectiveness and speed of adaptation.

Section 2 - System Requirements

2.1 System Overview

2.1.1 The system shall include an HTTP server capable of serving requests for monitoring, executing adaptations, and querying adaptation options for a self-adaptive exemplar.

2.1.2 It shall integrate with the UPISAS framework as an extension, enabling the exemplar to be tested and used within the UPISAS environment.

2.2 Functional Requirements

2.2.1 Monitoring Endpoint:

2.2.1.1 The system shall provide a /monitor GET endpoint.

2.2.1.2 The /monitor endpoint shall return a JSON object containing monitoring data for the exemplar.

2.2.2 Execution Endpoint:

2.2.2.1 The system shall provide an /execute PUT endpoint.

2.2.2.2 The /execute endpoint shall accept a JSON object as input to specify the adaptation to be executed.

2.2.2.3 The /execute endpoint shall trigger the specified adaptation process within the exemplar.

2.2.3 Adaptation Options Endpoint:

2.2.3.1 The system shall provide an /adaptation_options GET endpoint.

2.2.3.2 The /adaptation_options endpoint shall return a JSON object detailing available adaptation options for the exemplar.

2.2.4 Schema Endpoints:

2.2.4.1 The system shall provide a /monitor_schema GET endpoint to return the JSON schema of the monitor object.

2.2.4.2 The system shall provide an /execute_schema GET endpoint to return the JSON schema of the execute object.

2.2.4.3 The system shall provide an /adaptation_options_schema GET endpoint to return the JSON schema of the adaptation options object.

2.3 UPISAS Integration Requirements

2.3.1 Exemplar Module Implementation:

2.3.1.1 The developer shall implement an Exemplar class in a new .py file within the UPISAS/exemplars directory, named after the exemplar.

2.3.1.2 The new Exemplar class shall extend the existing Exemplar class from UPISAS.

2.3.1.3 The new Exemplar class shall implement the init() method.

2.3.1.4 The new Exemplar class shall implement the start_run() method.

2.3.2 Testing:

2.3.2.1 The developer shall verify the Exemplar class against the provided test cases.

2.3.2.2 All endpoints must pass the relevant test cases to ensure correct operation.

SECTION 3 - INSTALLATION AND SETUP

3.1 EWS:

3.1.1 Installation using DockerHub:

Installation and Configuration of the Emergent Web Server (EWS)

The project began with the team installing Docker. Following the instructions from the official Docker website, the team successfully installed Docker on all machines.

With the Docker environments prepared, the focus shifted to deploying EWS. We first wanted to try out installation through the DockerHub image. The following command was executed for this purpose:

```
docker run --name=ews -p 2011-2012:2011-2012 -d robertovrf/ews:1.0
```

This command pulled the EWS image and initiated a container instance named ews, mapping it to the designated ports.

We then verified the container's active status. Using the command below, we listed all running containers to confirm the presence and status of EWS:

```
docker container ls
```

After confirming that EWS was operational, the next step involved accessing the container's internal environment. We executed a command that connected us to the container's bash shell:

```
docker exec -it ews bash
```

Once inside the container, we we executed the interactive tool essential for starting EWS:

```
dana -sp ../repository InteractiveEmergentSys.o
```

The interactive tool's prompt appeared, and thus we were able to use the EWS environment to check if the provided interface was giving expected output.

3.1.2 Native Installation:

Following the successful Docker-based setup of EWS, we decided to proceed with a native installation, offering us the flexibility to modify and extend the artifact as needed. This process began by addressing the external software dependencies required for EWS.

The necessary software included the Dana programming language, Python 3.7, and Perl 5. We installed each, following the installation guides available on their respective official websites.

To compile the EWS project, the team utilized the 'make.dn' building tool located within the root directory of the cloned repository. The initial step was to compile 'make.dn' itself, followed by running it with the appropriate flags that matched our operating system: '-l' for Linux and MacOS and '-w' for Windows.

With 'make.dn' successfully compiled, we executed it to assemble the full EWS artifact:

```
dana make.o -l all
```

The compilation completed the codebase as required for running EWS properly.. The final step in the setup was to launch EWS using the EmergentSys.o component. By navigating to the emergent_web_server/pal/ directory and executing EmergentSys.o, we were able to start the web server:

```
dana -sp ../repository EmergentSys.o
```

Once the EWS was set up, the team was able to interact with its functionalities via the interactive tool inside the Docker container. We were able to query the system's current configuration, list all configurations, and initiate changes to the server's composition. Additionally, the interactive environment allows the team to monitor server performance metrics - response time, by sending requests through client scripts. The Perception module collects these performance metrics, and the Learning module uses them to determine the most efficient server configuration. The learning algorithm's parameters—observation window, exploration threshold, and number of rounds—can be adjusted to balance between accuracy and adaptability.

3.2 UPISAS

UPISAS which is used as an interface to run and test the working of EWS needed to be installed. After forking from the student branch of the original repository and cloning it onto the local machines, we first installed all the prerequisites as specified in the requirements text file by running:

```
pip install -r requirements.txt
```

3.3 CHALLENGES AND DESIGN DECISIONS:

Challenge: During the native install of EWS, team members with Apple Silicon M1 Chips encountered difficulties with the installation and setup of Dana.

After reading through the documentation of Dana, we figured out that the problem lies with the architecture of the processor - Dana was predominantly designed to work effectively on AMD64 processors and not on ARM64 processors.

Design Decision: We followed a hybrid approach. Figure 1 shows the docker compose file in EWS where we have clear instructions for MacOS users so that they would be able to run EWS by pulling the image directly from DockerHub whereas Linux or Windows users will be able to build the required docker images from the codebase directly.

```

services:
  ews:
    # ==== Comment out the next 4 lines for MacOS
    build:
      context: ../
      dockerfile: Docker/Dockerfile-ews
    image: ews-image
    # ====
    ports:
      - "8080-8081:2011-2012"
    # Uncomment the next line for MacOS
    # image: robertovrf/ews:1.0
    container_name: emergent_web_server

```

Figure 1: docker-compose file in EWS

Rationale: Even though natively building docker images from code was supposedly the best approach to go forward with, the hybrid approach was applied to improve collaboration among the team by making sure everyone is able to run the server on their systems using the same repository.

SECTION 4 - IMPLEMENTATION:

As part of the development, the team focused on implementing a series of RESTful endpoints essential for the operation of the Emergent Web Server (EWS). This implementation was carried out using FastAPI¹. The steps taken to implement each endpoint are as follows:

4.1 Monitoring Endpoint Implementation

4.1.1 The team initiated the FastAPI project setup, creating the main application file and defining the route for the /monitor endpoint, which was designed to handle GET requests.

4.1.2 We then implemented the logic to gather monitoring data from the EWS exemplar, ensuring the endpoint returned this data as a JSON object.

URL: http://emergent_web_server:2011/meta/get_perception

Note: 2011 is the docker port, they can only be accessed when Docker images are running.

¹ <https://fastapi.tiangolo.com/>

4.2 Execution Endpoint Implementation

4.2.1 Following the monitoring endpoint, the team developed the /execute endpoint, configuring it to respond to PUT requests.

4.2.2 Input validation was established to ensure the JSON object received contained the necessary information to specify an adaptation process. The Input accepts a JSON object consisting of ID and configuration. The input is consistent with the output we are getting from the /adaptation_options endpoint. In Swagger, the user is required to add the JSON object as a parameter to execute this endpoint.

4.2.3 We integrated the logic to get the config from the input object as the JSON input and trigger the corresponding adaptation process within the EWS exemplar.

4.2.4 The endpoint returns the message confirming the execution status of adaptation.

URL: http://emergent_web_server:2011/meta/set_config

Note: 2011 is the docker port, they can only be accessed when Docker images are running.

4.3 Adaptation Options Endpoint Implementation

4.3.1 The /adaptation_options endpoint was then set up to handle GET requests, aimed at providing clients with the various available adaptation options for EWS.

4.3.2 We populated the endpoint's response with a JSON object detailing the adaptation options, which included a dynamic retrieval of all configurations from the EWS exemplar.

4.3.3 The output is an array of all configurations where each element of the array is a JSON object consisting of a configuration and its respective ID.

URL: http://emergent_web_server:2011/meta/get_all_configs

Note: 2011 is the docker port, they can only be accessed when Docker images are running.

4.4 Schema Endpoints Implementation

4.4.1 Attention was shifted to the /monitor_schema endpoint, where the team implemented a method to serve the JSON schema defining the structure of the /monitor endpoint's response.

4.4.2 Similarly, for the /execute_schema endpoint, we provided a JSON schema response that clients could use to understand the required structure for the /execute endpoint's request payload.

4.4.3 Lastly, the /adaptation_options_schema endpoint was implemented to return the JSON schema associated with the adaptation options response format.

With these endpoints in place, the team had established a comprehensive API allowing for the monitoring, management, and adaptation of the EWS exemplar. Testing was conducted to ensure the endpoints behaved as expected and to validate the integration with the EWS exemplar's core functionalities.

4.5 Implementation of Exemplar Modules

As part of the integration with the UPISAS framework, the development team was tasked with implementing custom Exemplar classes. This was a critical step in enabling the Emergent Web Server (EWS) and the FastAPI server to function within the UPISAS environment.

4.5.1 EWS Exemplar Module

4.5.1.1 Implementation of EWS Exemplar Class: In alignment with the UPISAS structure, an EWS class was implemented within a new .py file in the UPISAS/exemplars directory. This class was designed to encapsulate the functionality of the EWS running as a Docker container.

4.5.1.2 Extension of Base Exemplar Class: The newly created EWS class extends the existing Exemplar base class from UPISAS, inheriting its properties and methods.

4.5.1.3 Initialization Method: The `__init__()` method of the EWS class was implemented to set up Docker configurations, such as container name, image, and port mappings. The `auto_start` parameter was included to determine whether the container should start immediately upon creation.

4.5.1.4 Start Run Method: The `start_run()` method contains the logic to execute the EWS using a series of commands within a tmux session, handling the initialization of the interactive system and the subsequent execution of client workload scripts.

```
def start_run(self):
    command1 = 'dana -sp ../repository InteractiveEmergentSys.o'
    command2 = 'cd ../ws_clients && dana ClientTextPattern.o'

    # Set sleep time for ews to start
    sleep_time = 120 # Increase if necessary (for MacOS)

    # Combine commands into a single tmux session
    combined_command = f'tmux new-session -d {command1} && sleep {sleep_time} && tmux send-keys -t 0 exit Enter && {command2}'

    self.exemplar_container.exec_run(cmd=["bash", "-c", combined_command], detach=True)
```

Figure 2: start-run method in your_exemplar.py

4.5.2 FastAPI Exemplar Module

4.5.2.1 Implementation of FastAPI Exemplar Class: Similarly, a FastAPI class was implemented to represent the FastAPI server as a separate Docker container. This class was also placed within the UPISAS/exemplars directory.

4.5.2.2 Extension and Initialization: Extending the same Exemplar base class, the FastAPI class's `__init__()` method sets up the necessary Docker configuration tailored for the FastAPI server, including container name, image, and port mappings.

4.5.2.3 Start Run Method: The `start_run()` method for the FastAPI class is prepared to contain any additional logic required for starting the FastAPI server, though currently, it passes as no further action is needed beyond the base class's startup procedures.

These implementations provide the foundation for the UPISAS framework to manage and interact with both the EWS and FastAPI servers, allowing them to be started, monitored, and controlled.

4.6 Challenges And Design Decisions:

Challenge: The team faced a technical challenge in automating the execution of command sequences required for EWS communication. The complexity arose from the need to run client scripts in a separate terminal session through a single command, which is not natively supported by standard command-line operations.

Design Decision: To address this challenge, we decided to leverage the capabilities of the `tmux` package. `tmux` allows for the management of multiple terminal sessions, enabling us to script the sequential execution of commands in an automated fashion.

Rationale: The use of `tmux` was driven by its ability to programmatically control terminal sessions, thus facilitating the automated setup of the EWS environment followed by the execution of client scripts. This approach allowed us to:

- Initiate the interactive EWS terminal and start up the server.
- Provide a sufficient wait period (sleep time) for the server startup process to complete.
- Exit the interactive terminal session post-initialization.
- Commence the execution of client scripts without manual intervention.

This method proved successful on our Linux test machine, though it required adjustments for the laptops with Apple M1 chips, which presented different timing requirements due to system-specific performance characteristics. To accommodate this variance, we included comments in our script guiding users on how to adjust the sleep time based on their operating system and performance considerations. This solution not only solved the immediate issue but also provided a flexible and robust approach for future adaptations and potential enhancements.

Design Decision: Use FastAPI to create the HTTP Server

Rationale: FastAPI was chosen for the HTTP server implementation due to its great performance, ease of asynchronous operations, and automatic documentation. It uses Python type hints which facilitates robust data validation and developer-friendly code. The framework's rapid development capabilities and extensive community support made it an ideal choice to meet our project's requirements while ensuring future adaptability. Moreover, FastAPI comes with automatically generated interactive API documentation accessible through the /docs endpoint. This Swagger UI feature allows for real-time visualization and direct interaction with the API's endpoints, offering a convenient and user-friendly way to test and debug server functionalities, which significantly aids in development and testing processes.

Design Decision: Exclude the pyEWS module from our project implementation.

Rationale: Upon reviewing our project's requirements, we determined that the functionalities offered by pyEWS were not necessary for the scope of our development. Direct interaction with the EWS's RESTful API adequately met our interaction needs, making the additional abstraction layer provided by pyEWS redundant. By omitting pyEWS, we simplified our codebase, reduced complexity, and minimized potential points of failure, resulting in a leaner, more maintainable system. This approach also aligns with our goal to streamline development and maintain system efficiency without the need for extra dependencies.

Challenge: The team encountered a significant obstacle when attempting to run the FastAPI server within the EWS Docker container. Despite various efforts, the server could not be successfully initialized and operated from within the same container as the EWS.

Design Decision: To overcome this challenge, it was decided to deploy the FastAPI server in a separate Docker container rather than within the EWS container.

Rationale: This separation into two distinct containers not only resolved the initialization issue but also adhered to the best practices of containerization, where each container runs a single process or service. By decoupling the two services, we enhanced the modularity and scalability of our system. To facilitate communication between the FastAPI server and EWS, a Docker network was created, enabling secure and efficient inter-container communication. This solution maintained the isolated environments, provided the necessary interconnectivity, and ensured that each service could be developed, scaled, and maintained independently.

SECTION 5 - TESTING

5.1 Implementation of Testing

5.1.1 Test Environment Setup

Initially, we focused on setting up the test environment. We started by instantiating instances of both EWS and FastAPI servers, with the containers set to launch automatically. This was critical to ensure that every test began with a fresh, running instance of each server.

Next, we established a dedicated Docker network, named `test_network`. This allowed for seamless communication between the EWS and FastAPI containers, mimicking the production environment they would eventually operate in.

5.1.2 Server Initialization Checks

Before proceeding with the test cases, it was essential to confirm that both servers were active and responsive. To achieve this, we implemented a waiting mechanism within the test setup that repeatedly attempted to connect to the servers at known endpoints. Only once a successful response was received did we proceed, ensuring that the servers were indeed ready for the testing procedures.

5.1.3 Conducting the Tests

With the servers confirmed operational, the team executed a suite of predefined test cases, to verify each functionality:

- We tested the adaptation options retrieval, confirming the server's ability to respond with the correct JSON data.
- Monitoring capabilities were checked to ensure operational metrics were being correctly reported.
- Execution tests were performed to validate the server's response to adaptation commands.
- Reachability tests for schema endpoints were conducted, ensuring the server provided the necessary JSON schemas.
- Moreover, we also tested by validating the JSON responses against their respective schemas.

5.1.4 Post-Testing Procedures

Upon completing the test cases, we stopped the containers and removed the Docker network programmatically. This cleanup was essential for maintaining the integrity of our testing environment, preventing residual states (i.e., running docker containers or unclosed network sockets) from influencing subsequent testing activities. Finally, the entire test suite was integrated into our pipeline.

Division of Work

Team Member	Work
[REDACTED]	Setting up EWS, FastAPI and UPISAS on Linux, Integrating and Testing
[REDACTED]	Setting up EWS and UPISAS on Mac, debugging and implementation of endpoints in HTTP Server and testing in UPISAS
[REDACTED]	Setting up of EWS and UPISAS, implementation of the API endpoints in HTTP Server. Check up of UPISAS functionality
[REDACTED]	Setting up EWS and UPISAS on Mac, configuring UPISAS, Drafting report