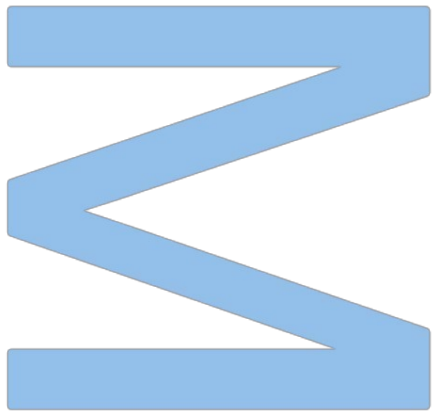
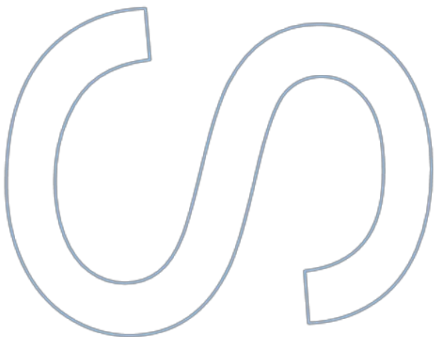
Execução de tarefas remotas a partir do CAOS





Henrique Vicente
Mestrado em Engenharia de Redes
e Sistemas Informáticos
Departamento de Ciências de Computadores
Faculdade de Ciências da Universidade do Porto
2025

Execução de tarefas remotas a partir do CAOS

Henrique Vicente

Dissertação realizada no âmbito do Mestrado
em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciências de Computadores
2025

Supervisor
José Proença, Professor Auxiliar, Faculdade de Ciências da
Universidade do Porto

Agradecimentos

Gostaria de expressar a minha sincera gratidão ao meu orientador, José Proença, por toda a orientação, disponibilidade e apoio fornecidos ao longo desta dissertação. Também agradeço à minha família pelo suporte, incentivo e compreensão em todos os momentos, e aos meus amigos, pela motivação e apoio constantes, que contribuiram para a concretização deste trabalho.

Resumo

O CAOS é uma *framework* e metodologia capaz de representar a estrutura de um dado modelo e de animar a sua semântica através de regras operacionais. Este tem como antecessor o Reolive, *framework* que analisa uma linguagem de coordenação de agentes via ferramentas complexas como verificadores de modelos, apresentando o seu resultado via interface *web*. No entanto, a implementação de novas análises no CAOS limita-se à utilização de Scala ou JavaScript e poderia ser útil a invocação de programas fora deste ambiente, tal como era possível no Reolive.

Esta dissertação estende o CAOS através do desenvolvimento de um servidor, que permite a execução remota de comandos do sistema operativo, um construtor que permite costumizar a interação entre o CAOS e o servidor, e *Process Runner*, programa em Scala que usa o CAOS e ilustra a funcionalidade desta extensão.

Palavras-chave: CAOS, Scala, http4s, servidor reutilizável, JavaScript *front-end*

Abstract

CAOS is a framework and methodology capable of representing the structure of a given model and animating its semantics through operational rules. Its predecessor is Reolive, a framework that analyzes an agent coordination language using complex tools such as model checkers, presenting its results via a web interface. However, the implementation of new analyses in CAOS is limited to the use of Scala or JavaScript, and it could be useful to invoke programs outside of this environment, as was possible in Reolive.

This dissertation extends CAOS through the development of a server that allows the remote execution of operating system commands, a builder that allows customizing the interaction between CAOS and the server, and Process Runner, a Scala program that uses CAOS and illustrates the functionality of this extension.

Keywords: CAOS, Scala, http4s, Reusable server, JavaScript front-end

Índice

Lista de Figuras

Lista de Abreviaturas

CAOS Computer-Aided design of structural Operational Semantics. iv, v, 1, 3– 5, 9, 15, 19, 22– 24, 27– 29, 31– 35

CSRF Cross-Site Request Forgery. 6

DSL Domain Specific Language. 8

JRE Java Runtime Environment. 5, 15

JVM Java Virtual Machine. 34

MVC Model-View-Controller. 5, 6

PR Process Runner. iv, v, 3, 9, 10, 12, 22– 24, 27, 31– 33

RAF RESTful API Framework. iv, 4– 8, 16

REST Representational State Transfer. iv, 5– 7, 19, 20, 34, 35

sbt Scala Build Tool. 5, 15

1. Introdução

O Computer-Aided design of structural Operational Semantics (CAOS) [1, 2] é uma *framework* e metodologia em desenvolvimento, cujo antecessor é o Reolive [3]. Ele fornece ferramentas para gerar um *website* constituído por elementos visuais e interativos com a representação da estrutura do modelo definido e animação da sua semântica através de regras operacionais, auxiliando o utilizador na compreensão do modelo em desenvolvimento.

Para ilustrar a interface do CAOS, foi realizada uma captura de ecrã (Figura 1.1) de um dos projetos desenvolvidos sobre o CAOS, RebeCaos [4]: uma *framework* utilizada para analisar e verificar modelos desenvolvidos na linguagem para modelos de atores Rebeca [5, 6].

Nesta figura, encontra-se a interface do RebeCaos é constituída por um conjunto de *widgets*, janelas visuais e interativas, cada uma com uma função diferente. Do lado esquerdo, encontram-se o “Input Rebeca program” (onde é inserido o programa), “Examples” (colapsado na imagem, consistindo numa coleção de possíveis programas exemplo) e o “View pretty data” (representação simplificada do programa). Do lado direito, estão presentes os elementos da interface do utilizador que facilitam a análise e verificação do programa.

Problema Atual Novas análises no CAOS têm de ser implementadas usando Scala ou JavaScript, sendo necessária a utilização de um *browser web* que suporte JavaScript para as correr. No entanto, em projetos como o RebeCaos, poderia ser útil utilizar um simulador nativo do Rebeca, como o Afra [7], via CAOS. Para além disso, o Reolive [3], predecessor do CAOS, inclui diversas análises que são realizadas remotamente, utilizadas em projetos como o Lince [8, 9], e que poderiam ser adaptadas para que utilizassem o CAOS. Por exemplo, poderia ser desenvolvido um servidor partilhado que auxilie na realização destas análises e possa ser utilizado por diversas pessoas em simultâneo.

Contribuições Em resposta a estas limitações, foquei-me em estender o CAOS de modo que novas análises possam ser realizadas através da invocação de simuladores e verificadores nativos em vez de ser necessária a sua implementação em Scala ou JavaScript. Para esse efeito, desenvolvi:

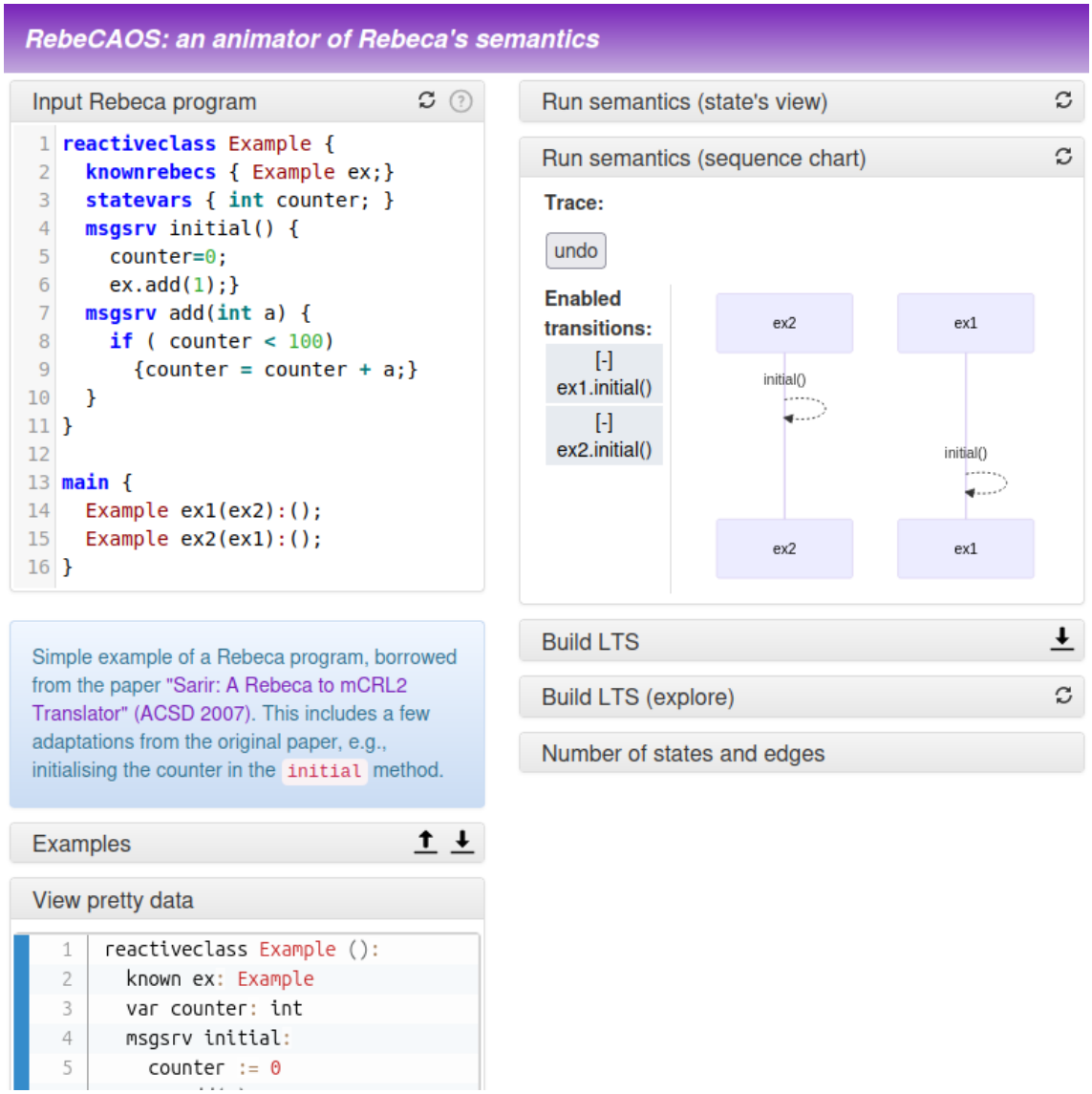


Figura 1.1: Captura de ecrã do RebeCaos.

1. um servidor que permite a execução de comandos do sistema operativo onde este se encontra a ser executado, com gestão de ciclo de vida de pedidos,

2. um construtor de *widgets* que permite receber respostas do servidor com o resultado de execuções de comandos requisitados e mostrando-o na interface do utilizador, e

3. um programa *Process Runner* que ilustra a utilização do *widget* e do servidor.

Com estas contribuições, deverá ser possível chamar o Afra [7] pelo RebeCaos, caso tenha suporte para linha de comandos. Nesta dissertação, o RebeCaos foi estendido com algo mais simples para ilustrar o processo: `wc`, comando utilizado para contar palavras e que se encontra disponível em sistemas Unix.

Exemplos Motivadores Para exemplificar a contribuição, foram utilizados dois projetos sobre o CAOS e que beneficiam da extensão descrita no decorrer da dissertação: **(1)** Process Runner (PR): um programa que desenvolvi para guiar o desenvolvimento do servidor e da estensão, e **(2)** RebeCaos [4], mencionado no início desta introdução.

Estrutura da dissertação Esta dissertação encontra-se dividida nos seguintes capítulos:

- **Introdução** (Capítulo 1): contextualiza o leitor sobre o estado atual do CAOS e existentes limitações. De forma semelhante, é explicitada a contribuição presente nesta dissertação, os exemplos motivacionais utilizados e a estrutura da mesma.
- **Estado de Arte** (Capítulo 2): descreve o Reolive, CAOS e restantes ferramentas usadas ao longo desta dissertação.
- **Exemplo motivador: PR** (Capítulo 3): explica o funcionamento do cliente utilizado, os padrões de comunicação, assim como o comportamento face a múltiplos clientes.
- **Servidor** (Capítulo 4): descrição do servidor mencionado nas contribuições. Inclui instruções de como o executar, descreve a sua implementação, é discutida a possível categorização do servidor quanto à arquitetura RESTful e é analisado como mostrar resultados parciais.
- **Cliente** (Capítulo 5): descrição da implementação realizada para estender o CAOS e a integrar no PR e RebeCaos.
- **Conclusão e trabalho futuro** (Capítulo 6): menciona as conclusões obtidas, bem como possíveis melhorias e extensões futuras.

2. Estado de Arte

Esta dissertação descreve como estender o CAOS, uma *framework* em Scala para analisar modelos, através do desenvolvimento de um servidor, que permite a execução remota de comandos do sistema operativo, e de um *widget*, que mostra na interface do utilizador o resultado da sua execução. Este capítulo descreve o Reolive (Secção 2.1), o CAOS (Secção 2.2) e o que é uma RESTful API Framework (RAF), assim como qual é utilizada (Secção 2.3).

2.1. Reolive

O Reolive [3], antecessor do CAOS, é uma *framework* desenvolvida em Scala em 2018 e que é executada num *browser web offline* capaz de compilar JavaScript. O seu objetivo inicial era converter um dado modelo que descreve a sincronização e coordenação de conectores na linguagem Reo em autómatos de portas e expressões algébricas que permitem ao verificador de modelos mCRL2 [10, 11] analisar e verificar esse modelo. Os resultados da sua análise e verificação são apresentados em elementos visuais e interativos que auxiliam os programadores a desenvolver modelos de coordenação de *software* confiáveis e eficientes.

Esta *framework* foi estendida de modo que a análise e verificação pudessem ser realizadas além do seu propósito inicial, abrangendo, entre outros, sistemas em tempo real [12], sistemas híbridos [8] e *Team Automata* [13]. Também foi desenvolvido um servidor em Scala através do Play, *framework* descrita na Subsecção 2.3.2, que, auxiliado por verificadores de modelos como o mCRL2 [10, 11] e o UPPAAL [14], assim como *constraint solvers*, suporta o funcionamento de *frameworks*, como o Lince [8, 9]. As diversas extensões implementadas no Reolive, algumas delas sem relação com Reo, resultaram na replicação de blocos de código e reutilização complicada. Por este motivo, no seu estado atual, o Reolive pode ser considerado monolítico, difícil de estender, manter e atualizar, resultando no desenvolvimento do CAOS, descrito na secção seguinte.

2.2. CAOS

O CAOS [1, 2] é uma *framework* e metodologia em Scala que se encontra em desenvolvimento e permite ao utilizador definir, analisar e testar um dado modelo. Mais concretamente, ele produz *websites* cuja interface de utilizador é constituída por elementos

visuais e interativos, gerados com auxílio de bibliotecas integradas, que apresentam a representação da estrutura do modelo e animam a sua semântica através de regras operacionais, permitindo ao utilizador detetar inconsistências, limitações e comportamentos inesperados. Projetos desenvolvidos sobre o CAOS necessitam de **Scala Build Tool (sbt)**, **Java JDK 11** (fornece o *Java Runtime Environment (JRE)* para compilar o projeto), e um ***browser web*** que suporte a compilação de JavaScript para ser executado. O código-fonte do CAOS, bem como alguns exemplos, pode ser encontrado em `https://github.com/arcalab/caos`.

A arquitetura do CAOS é baseada em três princípios: **(1)** o CAOS deve ser desenvolvido por meio de uma linguagem de programação de uso geral, facilitando a adoção e suportando *back-ends* mais complexos quando pretendido, **(2)** o resultado de projetos desenvolvidos sobre o CAOS deve ser intuitivo, sem a necessidade de plataformas ou instalações complexas, e **(3)** o CAOS deve ser fácil de reutilizar e projetos realizados sobre esta *framework* devem ser modulares e facilmente estendidos com novas análises. Mais detalhes sobre a sua arquitetura encontram-se descritos na Secção 5.1.

2.3. RAF

Um serviço *web* encontra-se bem desenvolvido quando é flexível, escalável, seguro e de fácil manutenção. Para garantir o bom desenvolvimento e funcionamento do meu servidor, foi utilizei uma RAF, *framework* que auxilia no cumprimento dos princípios Representational State Transfer (REST) (detalhados na Subsecção 2.3.1) e uso de arquiteturas como a Model-View-Controller (MVC), garantindo também que a comunicação cliente-servidor seja eficiente. Nas Subsecções 2.3.2 e 2.3.3 são apresentadas algumas alternativas à RAF Play (com uma comparação entre elas) e uma análise da que foi utilizada nesta dissertação, respetivamente.

2.3.1 Princípios REST

REST é um estilo de arquitetura que define padrões para o desenvolvimento de serviços *web* e que permite a comunicação entre sistemas diferentes, onde os clientes realizam pedidos através de métodos *HTTP*, como o *GET*, *POST*, *PUT* e *DELETE*, sendo retornada uma resposta num formato de dados específico (*JSON*, *XML* ou outro) [15]. Este estilo de arquitetura segue os 6 princípios mencionados abaixo [16].

- **Interface uniforme**: define uma interface consistente que identifica e, através de representações (normalmente, em formatos como *JSON* e *XML*), manipula cada um dos recursos. Os pedidos realizados pelo cliente devem conter a informação necessária de como devem ser processados e o cliente, através do *URI* inicial, pode interagir e/ou obter os recursos que pediu.
- ***Design* cliente-servidor**: o seu padrão de *design* reforça a separação entre a interface do cliente e armazenamento de dados do servidor, melhorando a portabilidade e a escalabilidade.
- **Sem estado**: cada solicitação deve conter toda a informação necessária de modo que as interações cliente-servidor não dependam do estado armazenado no servidor.
- **Armazenável em *cache***: a resposta do servidor, sempre que possível, deve ser armazenada em *cache*, uma vez que torna o serviço mais eficiente.
- **Sistema de camadas**: divide o sistema em camadas, onde cada uma tem uma determinada função, auxiliando no desenvolvimento e gestão do servidor.
- **Envio de código executável** (opcional): o servidor pode enviar código executável, na forma de *script*, extendendo a funcionalidade do cliente.

Um serviço é RESTful **apenas** quando todos os princípios, com exceção do último, são cumpridos.

2.3.2 RAFs em Scala

Play é uma *framework web* para Java e Scala com uma arquitetura MVC intuitiva e padrões de programação funcional. Inclui componentes essenciais que auxiliam no desenvolvimento de serviços RESTful eficientes e consistentes, sendo alguns deles servidor *HTTP* integrado, gestão de formulários, proteção contra Cross-Site Request Forgery (CSRF), bom sistema de rotas, suporte para diversos idiomas e *hot reloading*. Para além disso, o seu *design* sem estado não só usa Pekko para uma boa gestão de recursos, como também suporta o acesso a diversos tipos de bases de dados (MySQL, SQLite, entre outros). No entanto, apesar do Play ser uma boa *framework* para desenvolver serviços RESTful, tem diversas funcionalidades que não serão usadas para esta

Tabela 2.1: Listagem de RAFs consideradas.

RAF	**Descrição**	**Versões Scala suportadas**
Play	*Framework* sem estado e intuitiva que permite o rápido desenvolvimento de aplicações *web* eficientes, modernas e escaláveis. *Link: https://www.playframework.com*	2.13, 3.x.x
Finch	*Framework* desenvolvida sobre o Finagle utilizada para desenvolver serviços flexíveis, com bom desempenho e alta concorrência de forma intuitiva. *Link: https://finagle.github.io/finch*	2.12, 2.13
Scalatra	*Framework* simples e acessível, baseada no Sinatra, utilizada para o desenvolvimento de aplicações *web* escaláveis e eficientes. *Link: https://scalatra.org*	2.13, 3.x.x
http4s	*Framework* HTTP minimalista, tipada e funcional para Scala, com suporte para *streaming*. *Link: https://http4s.org*	2.13, 3.x.x, ScalaJS, Scala Native
Zio Http	*Framework* funcional construída em ZIO e Netty para criar aplicações *web* escaláveis, seguras e com alto desempenho. *Link: https://ziohttp.com*	2.13, 3.x.x

dissertação, consumindo espaço e recursos desnecessariamente. Como tal, foram exploradas possíveis alternativas a esta *framework*, encontrando-se listadas as principais na Tabela 2.1.

Para esta dissertação, o ideal é utilizar uma RAF minimalista, moderna e amplamente adotada, que garanta eficiência, intuitividade, escalabilidade e flexibilidade. O suporte a versões recentes do Scala também é importante para manter a compatibilidade com funcionalidades mais recentes. Para comparar as diversas alternativas ao Play, foi criada a Tabela 2.2, tendo como principais critérios estas características, refletindo a opinião pessoal do autor desta dissertação.

Como podemos observar na Tabela 2.2, das alternativas ao Play mencionadas anteriormente, só uma é que cumpre todos os critérios: **http4s**. Como tal, esta é a RAF que foi utilizada nesta dissertação. Na subsecção seguinte, são analisadas as suas principais características, vantagens e potenciais desafios que enfrenta.

Tabela 2.2: Comparação das alternativas ao Play.

Critério	**Finch**	**Scalatra**	**http4s**	**Zio Http**
Minimalista	Sim	Sim	Sim	Não
Moderno	Sim	Relativamente	Sim	Sim
Amplamente adotado	Não	Relativamente	Sim	Não
Eficiente	Sim	Sim	Sim	Sim
Intuitivo	Sim	Sim	Sim	Não
Oferece escalabilidade e flexibilidade	Sim	Sim	Sim	Sim
Suporta versões recentes do Scala	Não	Sim	Sim	Sim

2.3.3 RAF http4s

O http4s é uma *framework* puramente funcional e de tipo seguro para Scala que utiliza bibliotecas como Cats Effect e FS2, além de diversos *backends*, como Blaze, Ember e Jetty. Ela oferece uma abordagem intuitiva que suporta o desenvolvimento de aplicações *web* escaláveis, robustas e flexíveis, permitindo a declaração de rotas através da Domain Specific Language (DSL) http4s-dsl, garantindo transparência referencial e imutabilidade.

Esta *framework* também fornece suporte integrado para solicitações de *streaming*, permitindo o processamento de grandes quantidades de dados em tempo real, e interação com *APIs* externas de forma rigorosa e segura [17]. Para além disto, o http4s, através do Cats Effect, oferece controle de concorrência, segurança de recursos, uma gestão eficiente de memória e *threads* em aplicações de elevado desempenho. A sua boa integração com bibliotecas em Scala populares para a manipulação de *JSON*, autenticação e acesso a bases de dados permite também que o desenvolvimento de serviços *web* sustentáveis com o mínimo de requisitos [18].

Apesar das suas vantagens, o http4s enfrenta alguns desafios, especialmente para quem não se encontra familiarizado com programação funcional, uma vez que requer a compreensão de determinados conceitos como imutabilidade e *typeclasses*. Para além disto, tem mais *boilerplate* que outras RAFs, como o Play, o que dificulta o desenvolvimento de serviços, e o seu ecossistema, apesar de se encontrar em crescimento, não é tão extenso quanto o de outras RAFs, tornando a integração com aplicações e projetos externos mais desafiante.

3. Exemplo motivador: *Process Runner*

Este capítulo descreve *Process Runner* (PR), um programa que ilustra a funcionalidade da extensão do CAOS. Mais concretamente, é realizada uma análise à interface do PR e suas funcionalidades (Secção 3.1), seguindo-se a descrição dos padrões de comunicação entre o PR e o servidor (Secção 3.2), sendo, por fim, explicado o que ocorre caso existam múltiplos utilizadores (Secção 3.3).

3.1. Interface do *Process Runner*

O PR é um programa que consiste na reutilização do trabalho fornecido no enunciado do projeto da disciplina "Programação Concorrente". Na versão fornecida, incluia um parser e uma forma de comunicar com um servidor (não fornecido) de forma específica. A versão desenvolvida para esta dissertação foi modificada de forma a utilizar a nova funcionalidade da extensão do CAOS, nomeadamente através de *widgets* genéricos reutilizáveis criados para este efeito. A Figura 3.1 apresenta uma captura de ecrã da interface do PR e abaixo encontram-se descritos todos os seus componentes.

- **"Input program"**: recebe um programa que permite a execução remota de comandos do sistema operativo. Na Figura 3.1, o programa inserido ilustra como executar múltiplos comandos simultaneamente em servidores diferentes, existindo uma capacidade máxima que, quando atingida, coloca os restantes comandos solicitados em modo pendente até que o servidor tenha disponibilidade para os executar. O PR recebe o resultado da sua execução quando esta termina, mas quando o comando se encontra em modo pendente o utilizador só o recebe após algum tempo. É de notar que, apesar dos comandos serem executados pela ordem de receção do servidor, o PR recebe os resultados pelo tempo que demoraram a ser executados, em ordem crescente.

- **"Exampes"**: contém uma lista de programas exemplo que o utilizador pode inserir: execução de um ("ex1") ou mais comandos ("ex2") usando um servidor, existindo também a possibilidade de executar comandos usando múltiplos servidores ("ex3").

- **"Remote"**: *widget* desenvolvido para receber e apresentar o resultado da execução de cada comando na interface do utilizador. Contém também o botão "clear" que apaga o histórico de resultados obtidos até ao momento.

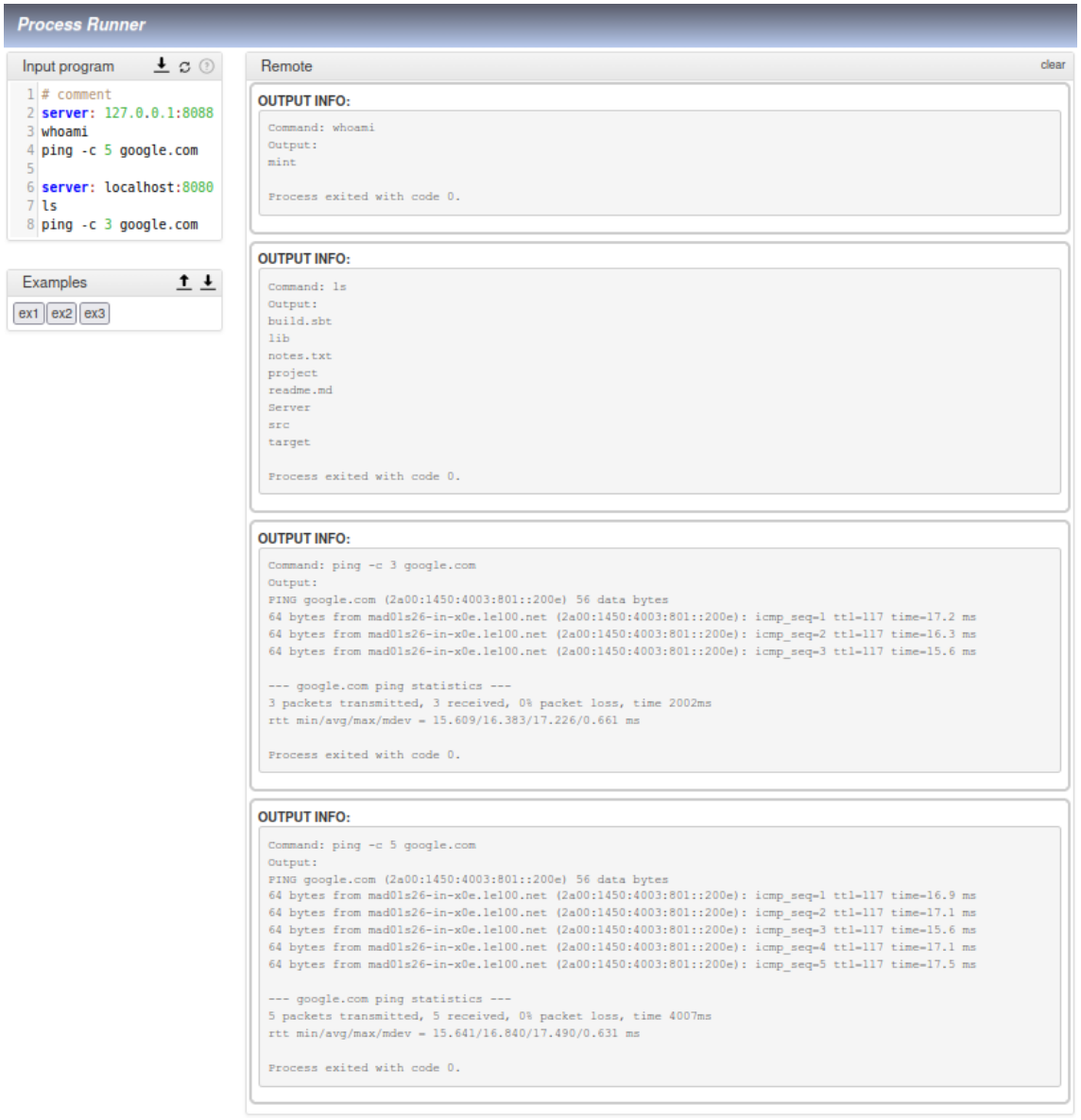


Figura 3.1: Captura da execução do PR.

Para uma melhor compreensão das suas funcionalidades, devem ser consultadas as Secções 3.2, 3.3 e 5.3, que apresentam a descrição das interações cliente-servidor existentes, do que ocorre quando existem múltiplos clientes e da implementação do PR, respetivamente.

3.2. Padrões de comunicação

Para o utilizador executar comandos remotamente com sucesso, é fundamental que a comunicação entre o cliente, PR, e o servidor seja eficaz e se encontre bem definida. Com o objetivo de ilustrar as interações existentes entre ambos, foi realizada uma análise do diagrama de sequência de cada um dos seus padrões de comunicação.

1. **Execução de comandos** (Figura 3.2):

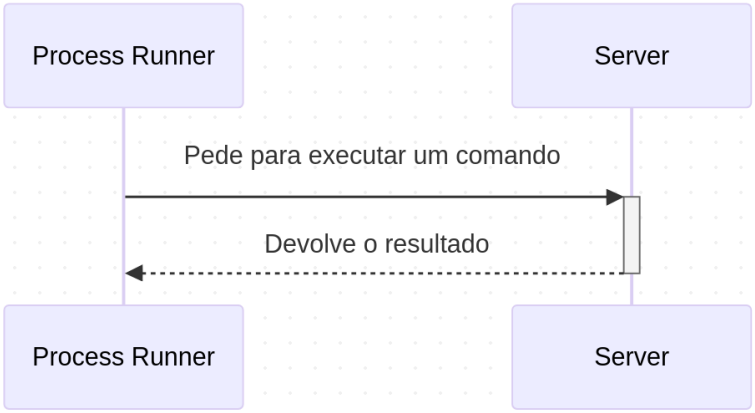


Figura 3.2: Diagrama de sequência da execução de um comando.

Para cada comando que o utilizador pretender executar, é realizado um pedido que contém o *token* do utilizador e o comando que será executado, sendo retornado o respetivo resultado após a sua execução. Relembrando a interface do PR (Figura 3.1), o utilizador solicita a execução de diversos comandos a dois servidores: "127.0.0.1:8088" e "localhost:8080", transformando o diagrama de sequência no apresentado na Figura 3.3. Em casos onde o utlizador solicita a execução de diversos comandos, a sua ordem de execução pode ser diferente da definida pelo programa, uma vez que o tempo que o servidor demora a receber cada uma das solicitações pode variar. É de notar que, apesar de nas Figuras 3.3 e 3.6 estar representado que o processamento das solicitações de execução de comandos é realizados por *threads*, na prática cada solicitação é processada por uma *fiber*, unidade de execução leve que permite elevada concorrência de forma eficiente e segura.

2. **Página de administradores** (Figura 3.4):

Quando um administrador pretende obter o estado global do servidor, é efetuado uma solicitação ao servidor e é retornada uma página *HTML* que não só apresenta o estado global do servidor como também fornece algumas funcionalidades que auxiliam a sua gestão e manutenção, garantindo o seu bom funcionamento. Mais detalhes sobre esta página encontram-se na Subsecção 4.3.3.

3. ***Reset* do servidor** (Figura 3.5):

Uma das funcionalidades fornecidas pela página de administradores é a realização de um *reset* ao servidor cuja solicitação, após processada, resulta no reinício do

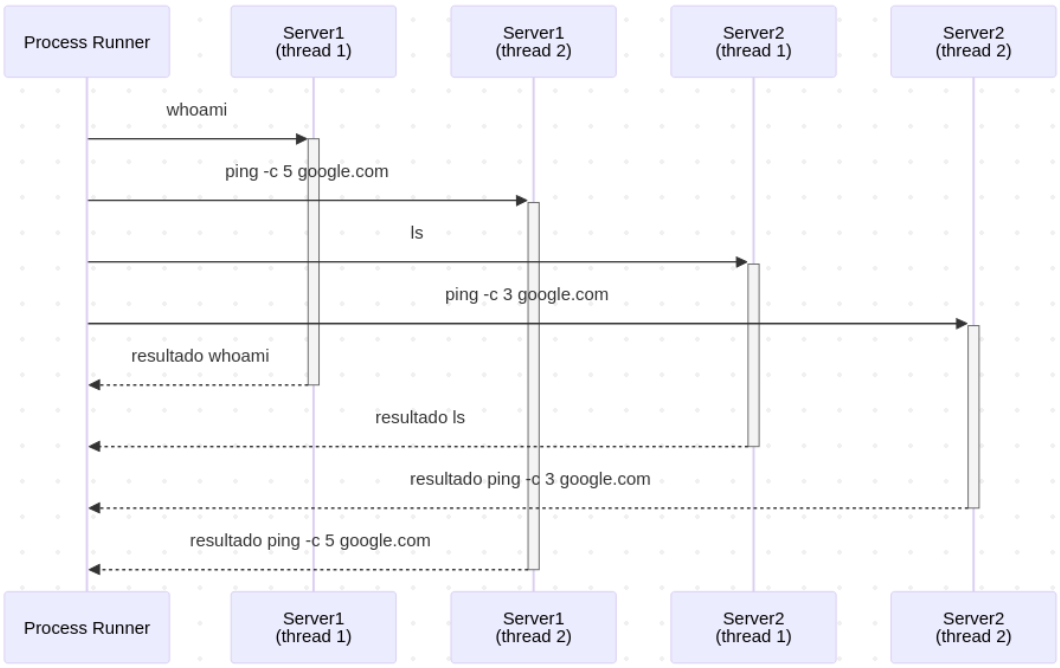


Figura 3.3: Diagrama de sequência da execução de diversos comandos usando mais do que um servidor.

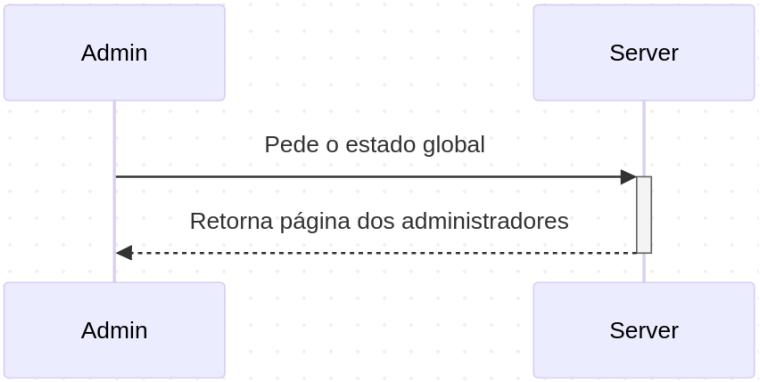


Figura 3.4: Diagrama de sequência da página dos administradores.

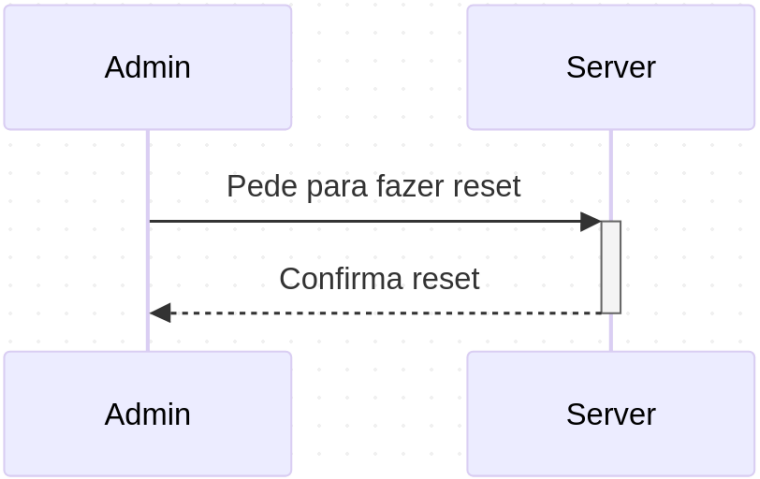


Figura 3.5: Diagrama de sequência do *reset* ao servidor.

estado do servidor, sendo retornada ao administrador uma confirmação de que tal ocorreu.

3.3. Múltiplos clientes

Na secção anterior, foram analisados os padrões de comunicação em cenários com apenas um utilizador a utilizar o PR, mas este programa pode ser utilizado por múltiplos utilizadores simultaneamente. Para ilustrar o que ocorre nestes casos, será usado um exemplo com três utilizadores.

Num caso onde existem múltiplos utilizadores a solicitar a execução de comandos, é essencial ter em consideração que estes serão executados em paralelo, tendo sido necessária a utilização de determinados mecanismos no desenvolvimento do servidor, como os descritos na Secção 4.2, que auxiliam na mitigação de problemas de comunicação e concorrência. A Figura 3.6 ilustra o diagrama de sequência de um possível caso onde três utilizadores tentam executar diversos comandos simultaneamente usando apenas um servidor.

Analisando o diagrama, podemos observar que, apesar de ser utilizado apenas um servidor, cada solicitação é processada individualmente pela respetiva *thread* (como mencionado anteriormente, encontra-se representado que as solicitações são processadas por *threads*, mas na prática o processamento é realizado por *fibers*). No entanto, é fundamental notar que:

- **A ordem de execução de comandos pode não ser cumprida**: a ordem da execução dos comandos pode ser diferente da definida no programa inserido dependendo do tempo que a solicitação demore a ser realizada e recebida pelo servidor, influenciado por diversos fatores, como a conexão de *internet*. Por exemplo, se o “Process Runner2” se encontrar com melhor conexão do que o “Process Runner1”, muito provavelmente o servidor receberá os pedidos do “Process Runner2” antes do que os do “Process Runner1”, mesmo que este tenha solicitado a execução dos seus comandos antes do “Process Runner2”.
- **O serviço pode apresentar alguma lentidão**: neste caso, o número de utilizadores é baixo, mas, se aumentarmos o seu número para 500, o servidor pode apresentar alguma lentidão no fornecimento do resultado da execução dos comandos. Isto acontece, não por causa da conexão da *internet*, mas sim pela quantidade de

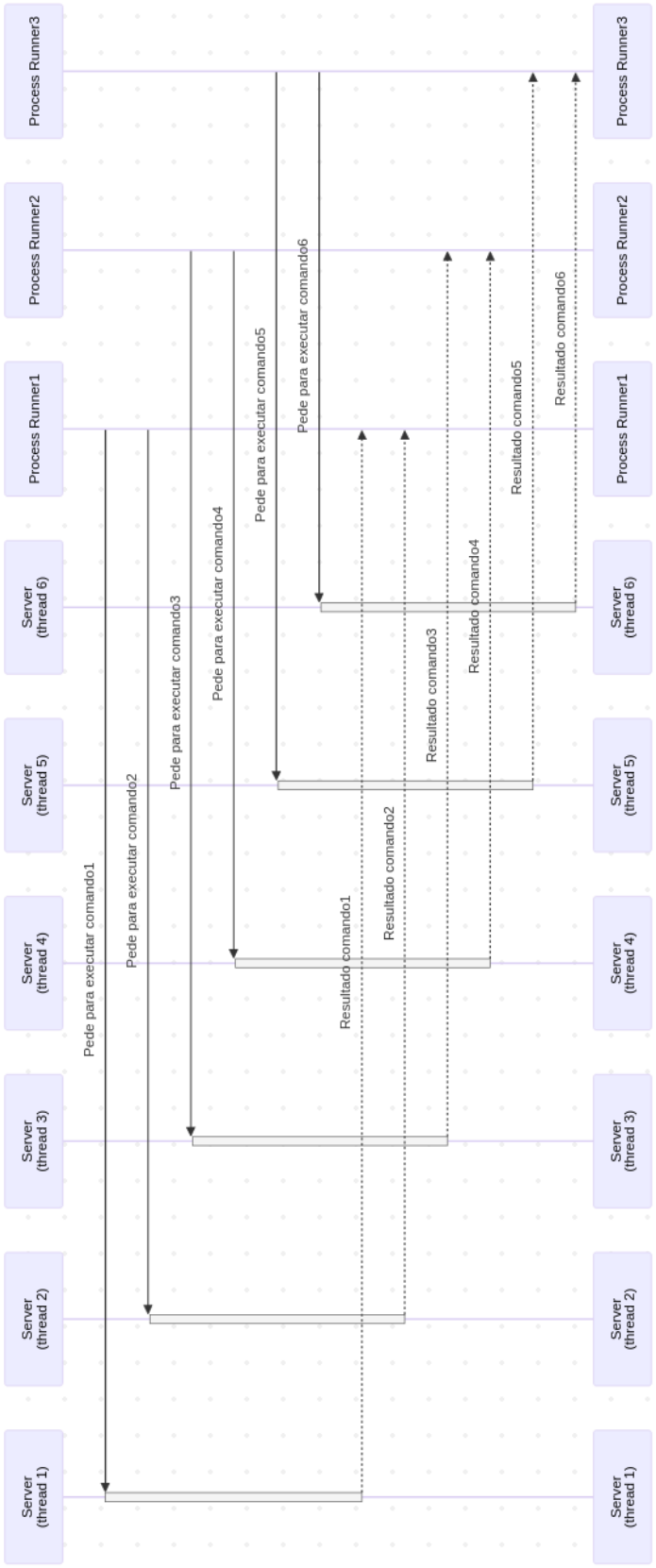


Figura 3.6: Diagrama de sequência da execução de comandos com diversos utilizadores.

comandos que o servidor consegue executar simultaneamente. Duas possíveis formas de mitigar este problema seriam o aumento desta quantidade, de modo que não afete o serviço negativamente, e a abordagem de um serviço distribuído, caso seja utilizado apenas um servidor. Esta segunda abordagem torna o serviço mais flexível, mas requer a utilização de um *load balancer* e de múltiplas máquinas virtuais, sendo recomendado a utilização de serviços *cloud* como o Google Cloud e a Amazon Web Services para o seu desenvolvimento e configuração.

4. Servidor

Este capítulo descreve o servidor desenvolvido que permite a execução de comandos do sistema operativo e acompanha a extensão do CAOS. Mais concretamente, são mencionados os requisitos e passos a seguir para o correr (Secção 4.1), seguindo-se a explicação das estruturas de dados partilhadas utilizadas no seu desenvolvimento (Secção 4.2) e a análise detalhada das rotas, arquitetura e implementação do servidor (Secção 4.3). Também é aberta uma discussão que verifica se o servidor é RESTful (Secção 4.4), sendo por fim mencionadas possíveis abordagens que permitem mostrar o resultado de uma forma parcial (Secção 4.5).

4.1. Como correr o servidor?

Antes de correr o servidor, é essencial garantir a instalação de duas depedências: **(1)** sbt, ferramenta que permite a transferência das bibliotecas necessárias, compilar e testar diversos projetos, e **(2)** Java JDK 11*, que fornece o JRE necessário para o funcionamento do sbt.

Para correr servidor, o utilizador deve clonar o projeto do GitHub, que se encontra em `https://github.com/RaidenEi2020/Server.git`, e na pasta do projeto, abrir o terminal e executar os comandos `sbt compile`, para transferir as bibliotecas e compilar o projeto, e `sbt run` para correr o servidor.

4.2. Estruturas de dados partilhadas

No servidor para que o utilizador consiga executar diversos comandos simultaneamente é necessário o suporte de concorrência e execução de tarefas, neste caso execução de comandos do sistema operativo, em paralelo, a qual envolve a manipulação de um estado partilhado. Para que tal suporte fosse implementado, foram usados os mecanismos:

- **Atomic Integer**: valor inteiro que é atualizado de forma atómica, garantido que operações como aumento ou diminuição do valor sejam efetuadas sem problemas de concorrência.
- **Linked Blocking Queue**: estrutura de dados que consiste numa *queue* onde, caso uma operação esteja a ser realizada, bloqueia a realização de outras, colocando-as

*Pode funcionar com versões do Java JDK mais recentes, mas não foi testado.

em espera até que a atual termine. No meu servidor, existem 3 `LinkedBlocking Queues`: `runningProcesses`, que contém os processos que se encontram em execução, `pendingQueue`, constituída pelos que se encontram em estado pendente, aguardando a sua execução; e a `processHistory`, que guarda o histórico dos comandos já executados.

- **Semaphore**: estrutura utilizada para limitar a quantidade de processos que podem ser executados em simultâneo. Caso o limite seja atingido, a execução de outros comandos é bloqueada até que o servidor termine a execução de pelo menos um dos que se encontram em execução. Quando tal ocorrer, ele desperta um ou mais processos, consoante a capacidade disponível, através de um `Signal`, respeitando a ordem de chegada das solicitações de execução.

4.3. Detalhes de implementação

Uma vez explicadas as estruturas de dados utilizadas usadas e qual o propósito de cada uma delas, são analisadas a arquitetura e implementação usadas (Subsecção 4.3.1). As rotas `/run-process` e `/admin-panel` encontram-se descritas nas Subsecções 4.3.2 e 4.3.3, respetivamente, uma vez que representam um papel importante na interação do cliente com o servidor.

4.3.1 Arquitetura e implementação

Como mencionado anteriormente, foi utilizada a RAF http4s no desenvolvimento e configuração do servidor, uma vez que permite a sua realização de forma intuitiva e eficiente. A sua estrutura encontra-se na Figura 4.1.

Nesta, podemos observar que o servidor é constituído pelo *package* `utils`, onde se encontra o `ProcessManager.scala` e o `ServerStatus.scala`, e três ficheiros: `Main.scala`, `Server.scala` e `Routes.scala`, contendo cada um o respetivo objeto. O **ProcessManager** é quem cria, gere os processos que são executados e define quantos o servidor pode executar em simultâneo. Este objeto é constituído pelas funções:

- `waitAndRunProcess(token: String, cmd: String): IO[Response[IO]]`: gera o *id* do processo e armazena no `process`, uma classe `ProcessData(id: Int, token: String, cmd: String, output: Option[String])`, a informação importante sobre o processo: *id*, *token* do cliente, comando e o respetivo resultado. Após a geração do processo de execução do comando solicitado, ele é adicionado à

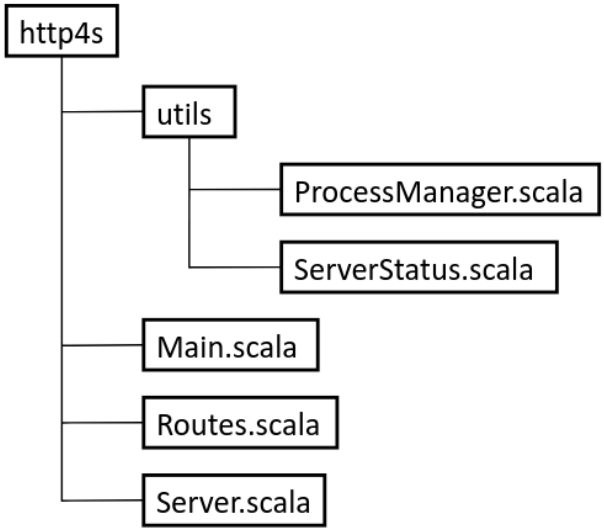


Figura 4.1: Estrutura do servidor.

`pendingQueue`. O processo permanece nesta *queue* até que o `semaphore` verifique que ele é o primeiro elemento da lista e o servidor tenha capacidade para o executar. Caso possa ser executado, é removido da `pendingQueue` e o comando do respetivo processo é executado, retornando o resultado da função `runProcess`.

- `runProcess(process: ProcessData): IO[Response[IO]]`: é executada dentro da função `waitAndRunProcess` e recebe como argumento o `process` lá gerado. Executa o comando recebido e o resultado gerado no terminal é guardado no `outputBuilder` (como o `ProcessLogger` é uma estrutura que processa cada linha do resultado individualmente, foi necessário o uso deste `StringBuilder`). Quando a execução de um comando começa, os respetivos *id* e processo são adicionados à `runningProcesses`, mas quando a sua execução termina são removidos e é adicionado à `processHistory` a informação do processo executado. Após a conclusão destas tarefas, é retornada uma resposta com o *id*, comando, *token* do utilizador e resultado do comando executado.

- `resetAll: Unit`: realiza um *reset* ao servidor, que consiste em parar todos os processos em execução e remover todos os elementos da `pendingQueue`, `runningProcesses` e `processHistory`.

O **ServerStatus** fornece a informação necessária para a equipa de administradores sobre o estado global do servidor, que apresenta o tempo de atividade (`getUptime(): String`), memória utilizada (`getMemoryUsage(): String`), número de *threads* em uso (`getThreadCount(): Int`), carga do servidor (`getSystemLoad(): String`) e o número

de processos em espera (`getQueueSize(): Int`). O objeto **Main** corre o servidor usando as configurações definidas no **Server**. O **Routes** é define as rotas do serviço e o respetivo comportamento, sendo estas **(1)** `/run-process` (Subsecção 4.3.2), que permite a execução de comandos e retorna o seu resultado, **(2)** `/admin-panel` (Subsecção 4.3.3), para gerar a página *HTML* dedicada à equipa de administradores, e **(3)** `/reset`, utilizada para realizar o *reset* do servidor através da função `resetAll`, retornando uma confirmação do sucesso ou falha desta operação.

4.3.2 Rota /run-process

A rota `/run-process` é a mais importante que o servidor possui, dado que permite a execução remota de comandos do sistema operativo de forma intuitiva, sendo necessário o utilizador fornecer o comando que pretende executar e o seu *token* como argumentos no *URL*, tornando este `/run-process?cmd=<commando>&token=<token>`. Caso não seja fornecido um comando, o resultado do acesso a esta rota é um erro a informar que não foi apresentado nenhum comando para executar, mas se apenas o *token* não for passado como argumento, o comando é executado e o valor do *token* é “unknown”.

4.3.3 Rota /admin-panel

Para auxiliar os administradores na gestão do servidor e assegurar o seu funcionamento, foi implementada uma rota específica: `/admin-panel`. Esta, quando acedida, retorna a página *HTML* ilustrada na Figura 4.2 que fornece ferramentas que permitem ao administrador ter uma visão global do estado do servidor, executar comandos e, caso seja necessário, realizar um *reset*. Como podemos observar, esta página é constituída pelos seguintes componentes:

- **Botão “Start New Process”**: botão que permite ao administrador a execução remota de comandos do sistema operativo. Quando clicado, apresenta uma janela que permite ao administrador inserir o comando que pretende execução e, após a sua submissão, acede à rota `/run-process` com o *URL* `/run-process?cmd=<comando>&token=admin`. Quando a sua execução terminar, é apresenta outra janela com o seu resultado. As solicitações de execução de comandos realizadas pelos administradores não têm prioridade sobre as realizadas pelos clientes.

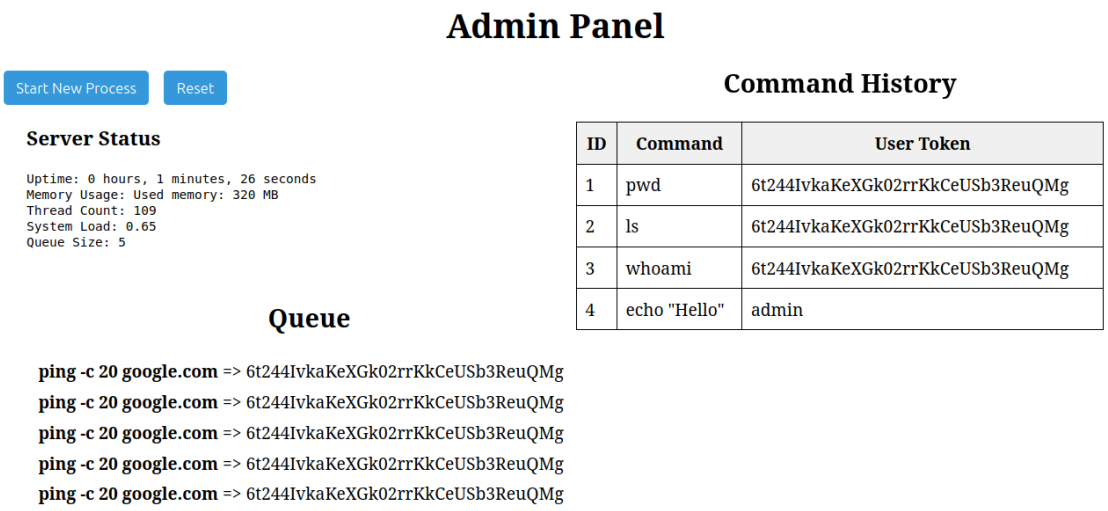


Figura 4.2: Página dos administradores.

- **Botão "Reset"**: quando clicado, apresenta uma janela que requisita a confirmação da realização do *reset* ao servidor e, caso o administrador confirme, rota `/reset` será acedida , retornando uma mensagem a confirmar que fez o *reset*.
- **Secção "Server Status"**: secção com as informações obtidas pelo objeto `Server` para fornecer ao administrador uma visão do estado global do servidor.
- **Secção "Queue"**: lista todos os processos de execução de comandos que se encontram em estado pendente, bem como o *token* do respetivo utilizador que solicitou a sua execução. Mostrar o *token* do utilizador tem como objetivo distinguir quem realizou a solicitação de execução de um ou mais comandos, onde "admin" é o *token* dos administradores e os outros são de clientes que utilizam a extensão do CAOS.
- **Tabela "Command History"**: tabela que fornece ao administrador o histórico completo dos comandos já executados, onde cada linha contém o *id*, comando e *token* do utilizador.

4.4. Discussão: o servidor é RESTful?

Ao longo da análise da arquitetura e implementação do servidor, torna-se pertinente verificar se o servidor respeita os princípios REST. Como tal, foi criada a Tabela 4.1, que apresenta uma revisão destes princípos e indica se são ou não respeitados.

Como se pode concluir pela análise da Tabela 4.1, o servidor, apesar de ter sido implementado com base nos principios REST, não é completamente RESTful, uma vez que não os cumpre todos.

Tabela 4.1: Avaliação do servidor face aos princípios REST.

Principio REST	**Verificação**
Interface uniforme: define uma interface consistente que identifica e, através de representações (normalmente, em formatos como *JSON* e *XML*), manipula cada um dos recursos. Os pedidos realizados pelo cliente devem conter a informação necessária de como devem ser processados e o cliente, através do *URI* inicial, pode interagir e/ou obter os recursos que pediu.	O servidor fornece uma interface consistente que, a partir de um único *URI* (rota `/admin-panel` para os administradores e `/run-process` para os restantes utilizadores), é possível interagir com os seus recursos, que neste caso consistem em solicitar a execução de um ou mais comandos e fornecimento do seu resultado, onde em cada solicitação é fornecida a informação necessária através dos argumentos do *URL*.
***Design* cliente-servidor**: o seu padrão de *design* reforça a separação entre a interface do cliente e armazenamento de dados do servidor, melhorando a portabilidade e a escalabilidade.	Relembrando a estrutura do servidor ilustrada na Figura 4.1, ele foi implementado de modo que o objeto `ProcessManager` realize a gestão dos processos, armazenando os dados necessários, e a interface do utilizador de cada uma das rotas se encontre definida no `Routes`.
Sem estado: cada solicitação deve conter toda a informação necessária de modo que as interações cliente-servidor não dependam do estado armazenado no servidor.	O servidor contém estado interno que permite ao `ProcessManager` gerir os processos, mas as interações cliente-servidor existentes não dependem desse estado e cada solicitação contém a informação necessária para a sua realização (os adminstradores acedem à rota `/admin-panel` sem argumentos no *URL* e os restantes utilizadores acedem à rota `/run-process` com os argumentos `cmd` e `token`).
Armazenável em *cache*: a resposta do servidor, sempre que possível, deve ser armazenada em *cache*, uma vez que torna o serviço mais eficiente.	No seu estado atual, o servidor não utiliza *cache* para tornar o serviço mais eficiente, mas em versões futuras poderá existir uma versão mais específica com esta otimização.
Sistema de camadas: divide o sistema em camadas, onde cada uma tem uma determinada função, auxiliando no desenvolvimento e gestão do servidor.	Relembrando a estrutura do servidor, ilustrada na Figura 4.1, esta encontra-se dividida em três partes: infraestrutura, rotas e lógica. A infraestrutura (`Server.scala` e `Main.scala`) contém as configurações necessárias para o servidor correr, a "rotas" (`Routes.scala`) é constituída pela definição das rotas e respetivas interações, e a lógica (*package* `utils`) define a lógica do servidor, ou seja, o que ocorre em cada uma das interações, como os processos correm e são geridos.
Envio de código executável (opcional): o servidor pode enviar código executável, na forma de *script*, extendendo a funcionalidade do cliente.	As respostas fornecidas pelo servidor não consistem em código executável, mas sim numa `String` que mostra ao utilizador o resultado da execução do comando solicitado.

4.5. Como mostrar resultados parciais?

Quando um utilizador realiza uma solicitação de execução de um comando, o servidor executa-o e fornece-lhe uma resposta com o seu resultado completo, mas em versões futuras pode existir a necessidade de mostrar esses resultados de forma parcial, mas para tal, é necessário que, algum tempo após o começo da execução de um comando, o servidor envie uma resposta com o resultado obtido até ao momento e informe do estado da sua execução, fornecendo também a possibilidade de o utilizador receber a restante parte do resulta de forma parcial ou por completo, dependendo do caso.

Uma possível abordagem para implementar este suporte é a utilização de funções como `IO.race` para verificar se a execução de um comando termina antes de um tempo estabelecido e, caso não termine, a sua execução continuaria e o servidor enviava ao servidor o resultado obtido até ao momento. Para o utilizador receber a restante parte do seu resultado, poderia ser implementado por exemplo um botão que permite a atualização do resultado, fornecendo também o estado da execução do comando solicitado.

5. Cliente

Este capítulo descreve a implementação realizada para estender o CAOS. Mais concretamente, são analisadas as arquiteturas do CAOS e do PR, assim como a sua conexão (Secção 5.1), seguindo-se os detalhes de implementação da extensão do CAOS (Secção 5.2) e do PR (Secção 5.3). Por fim, é descrito como integrar a sua extensão no RebeCaos (Secção 5.4).

5.1. Arquitetura do CAOS e do PR

Antes de descrever a implementação realizada, são analisadas as arquiteturas do CAOS e do PR com o objetivo de ilustrar a conexão entre o CAOS e o cliente que o utiliza. Esta secção descreve a parte da arquitetura do CAOS relevante para este trabalho.

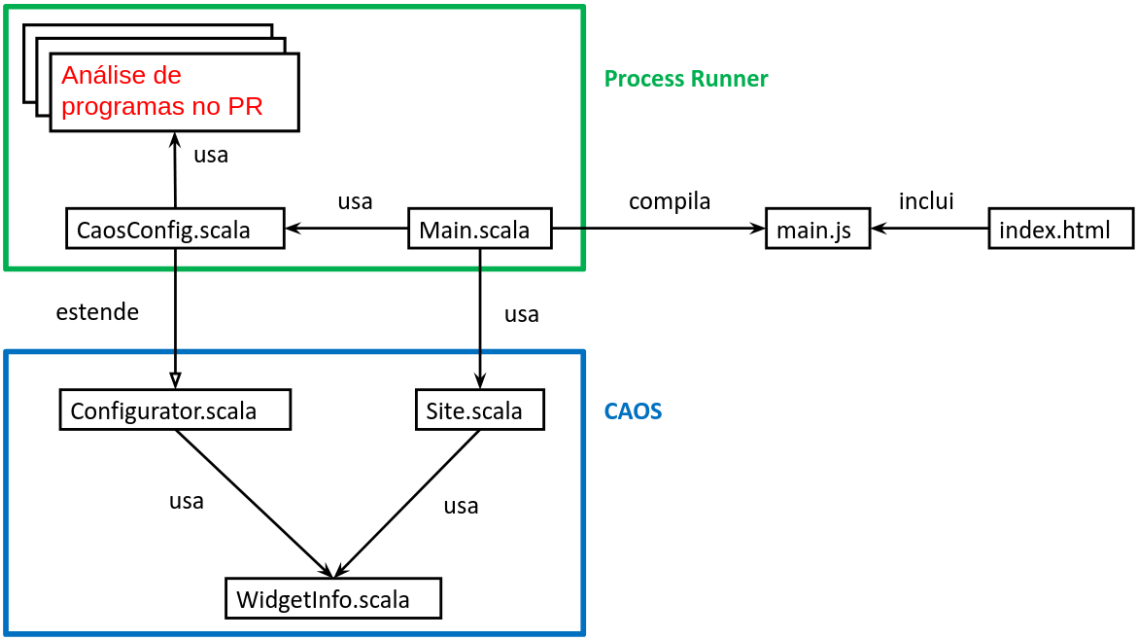


Figura 5.1: Arquitetura do CAOS e do PR.

A Figura 5.1 ilustra a arquitetura do CAOS e do PR, assim como a sua conexão e o seu resultado. O **PR** permite a compilação e geração do *website* do CAOS, sendo necessário abrir o ficheiro `index.html` num *browser web* para o correr. Ele é constituído pelo *package* `frontend` e contém dois ficheiros:

- `CaosConfig.scala`: configuração que estende o `Configurator.scala` e especifica qual o *parser* utilizado para analisar os programas inseridos, os exemplos disponíveis e os *widgets* que constituem o *website* gerado.

- `Main.scala`: que utiliza esta configuração e a lógica definida pelo `Site.scala` para gerar o conjunto de ficheiros que constituem o *website*.

A "Análise de programas no PR" mencionada na Figura 5.1 representa o conjunto de ficheiros responsáveis pela análise e processamento dos programas inseridos. Neste caso e relembrando a sua interface, ilustrada na Figura 3.1, após o programa ser inserido no *widget* "Input program", é analisado e é gerado um `Map` cuja chave e valor são, respetivamente, o servidor que será utilizado e o conjunto de comandos que serão executados. Este `Map` será usado para preparar as solicitações que serão enviadas para o servidor.

A Figura 5.2 ilustra a estrutura do **CAOS** no seu estado atual e informa que ficheiros foram modificados e adicionados.

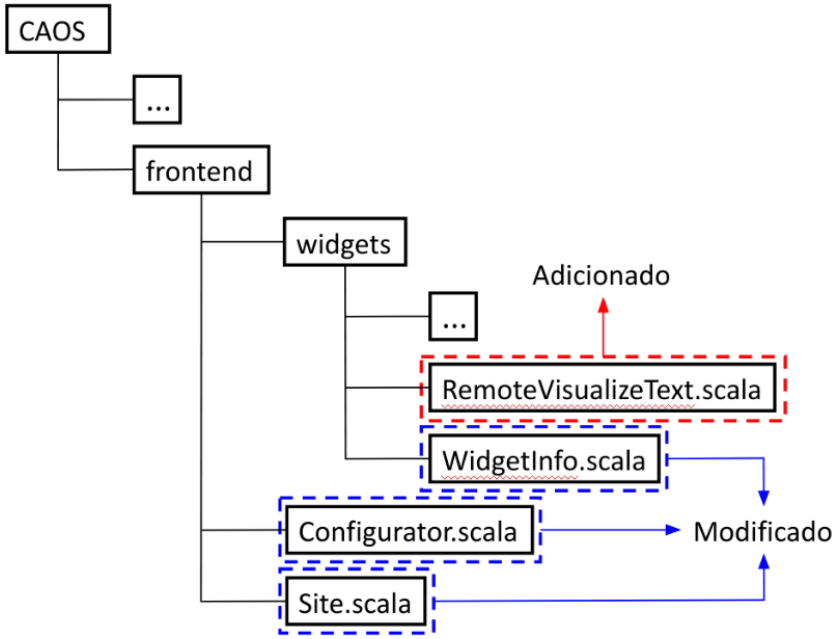


Figura 5.2: Estrutura do CAOS.

Um dos *packages* que constituem o CAOS é o "frontend", utilizado para definir os elementos e respetivas interações do *website*. Como se pode observar na Figura 5.2, este *package* é constitído por:

- **"Widgets"**: contém o conjunto de ficheiros com os construtores de *widgets*, responsáveis por gerar os *widgets*, seus elementos e respetivas interações, e a assinatura de cada um deles, definidas no `WidgetInfo.scala`.

- `Configurator.scala`: define a estrutura abstrata da interface do *website*, que inclui a definição do seu nome, *parser*, exemplos e como o `CaosConfig.scala` deve chamar cada uma das assinaturas dos *widgets* na sua configuração.
- `Site.scala`: estabelece a lógica do *website*, ou seja, inicializa a sua interface, organiza os seus elementos visuais, gere os exemplos e construtores de *widgets* utilizados na inicialização.

Nas Secções 5.2 e 5.3, encontram-se descritas as implementações da extensão do CAOS e do PR, respetivamente, com o objetivo de melhorar a compreensão da lógica dos blocos que os representam na Figura 5.1.

5.2. Estender o CAOS: implementação

Para estender o CAOS, foi desenvolvido o *widget* "Remote" que, dada uma sintaxe, possibilita ao utilizador a execução de um conjunto de comandos do sistema operativo e mostrar o seu resultado na interface do utilizador. Relembrando a arquitetura do CAOS e do PR, descritas na secção anterior, enumeramos os passos necessários para a implementação deste novo *widget*.* O código desta extensão pode ser encontrado em `https://github.com/RaidenEi2020/caos`.

1. **Adicionar o *widget* à estrutura do *website***: no ficheiro `Configurator.scala`, foi adicionada a função `viewRemote[Stx]`, que permite ao PR usar o novo *widget*, retornando a chamada da respetiva assinatura, `VisualizeRemote[Stx]`, descrita no passo seguinte.

```
def viewRemote[Stx](buildCommands: Stx => List[(String, String)],
    generateHtml: String => String, remember: Boolean = false): WidgetInfo[Stx] =
    VisualizeRemote[Stx](buildCommands, generateHtml, remember)
```

   Nesta nova função definida, encontram-se os argumentos:

   - `buildCommands: Stx => List[(String, String)]`: função que, dada uma sintaxe, gera os comandos e retorna uma lista de pares `(String, String)`, onde o primeiro elemento é o *ip* do servidor e o segundo o comando solicitado.
   - `generateHtml: String => String`: utiliza a resposta fornecida pelo servidor para gerar e retornar código *HTML* que permite ao utilizador visualizar o resultado da execução do respetivo comando.

*Os passos descritos foram os utilizados para o PR, mas, ao integrar esta extensão com outras ferramentas, poderão existir alterações, como é o caso do RebeCaos, descrito na Secção 5.4

- remember: Boolean: variável que, se for verdadeira, faz com que o *widget* guarde o resultado da execução dos comandos solicitados anteriormente, mas se for falsa, mostra apenas o resultado dos comandos solicitados no programa inserido. Se não for fornecida, é aplicado o caso em que ela é falsa.

2. **Definir a sua assinatura**: no `WidgetInfo.scala`, foi implementada a assinatura do novo *widget*, `VisualizeRemote[Stx]`, classe genérica que estende `WidgetInfo[Stx]`, um tipo abstrato que estabelece a base dos *widgets*. Os argumentos utilizados na sua definição são idênticos ao descritos no passo anterior.

```
case class VisualizeRemote[Stx](buildCommands:Stx => List[(String, String)],
    generateHtml: String => String, remember: Boolean)
    extends WidgetInfo[Stx]
```

3. **Atualizar a lógica do *website***: tendo sido definida a sua assinatura, é necessário garantir que o construtor do *widget* "Remote", `RemoteVisualizeText[Stx]`, descrito no passo seguinte, é inicializado sempre que a sua assinatura é chamada e, para tal, o ficheiro `Site.scala` foi modificado de modo que na função `mkWidget`, responsável por associar as chamadas das assinaturas dos *widgets* à inicialização dos respetivos construtores, este novo caso se encontre presente.

```
protected def mkWidget[Stx](w: (String,WidgetInfo[Stx]), get:()=>Stx,
    getAll:()=>Seq[(String,Stx)], out:OutputArea, doc: Documentation): Widget[Unit] =
    try w._2 match {
            ...
                case VisualizeRemote(buildCommands, generateHtml, remember) =>
                    new RemoteVisualizeText(()=>buildCommands(get()), generateHtml,
                    remember, w._1,out,doc)
                ...
        }
```

Os argumentos utilizados para adicionar este novo caso são quase todos idênticos aos descritos no primeiro passo, com exceção do `()=>buildCommands(get())`, `w._1`, `out` e `doc`. Este primeiro é um argumento *lazy*, ou seja, o seu valor só será calculado quando a função for executada e consiste no resultado da execução da função `buildCommands`, utilizando a `get` como seu argumento para obter a sintaxe resultante da análise do programa inserido. O `w._1`, `out` e `doc` contêm o nome, caixa com os erros obtidos e documentação do *widget*, respetivamente.

4. **Implementação do *widget* "Remote"**: foi adicionado ao *package* `widgets`, o ficheiro `RemoteVisualizeText.scala`, com a respetiva classe, que contém o construtor deste *widget*, os elementos visuais que gera e as respetivas interações.

```
class RemoteVisualizeText[Stx](cmds:()=>List[(String,String)],
    generateHtml: String => String, remember: Boolean, name:String,
    errorBox: OutputArea, doc:Documentation)
    extends Widget[Unit](name,doc)
```

Os argumentos `generateHtml` e `remember` são idênticos aos descritos no primeiro passo, porém, o `cmds:()=>List[(String,String)]` é *lazy* e retorna a lista de pares, onde o primeiro elemento é o *ip* do servidor e o segundo o comando solicitado, o `name:String`, `errorBox: OutputArea` e `doc:Documentation` consistem no nome do *widget*, caixa que mostra os erros ocorridos e a sua documentação, respetivamente.

Na classe deste novo construtor, foram utilizadas as seguintes funções para definir os elementos visuais e as respetivas interações:

- `init(div: Block, visible: Boolean): Unit`: inicializa o construtor do *widget* "Remote" e gera os seus elementos visuais: botão "Clear", para apagar o histórico dos resultados da execução dos comandos solicitados pelo utilizador obtidos, e onde se encontram apresentados na interface.
- `update(): Unit`: verifica se o *widget* se encontra exibido e, se estiver, o utilizador consegue executar comandos remotamente e visualizar os seus resultados.
- `generateToken(length: Int): String`: utilizada na geração do *token* de cada utilizador.
- `clear: Unit`: é utilizada pelo botão "Clear" para apagar o histórico dos resultados obtidos.
- `buildUrl: String`: para cada comando, gera o respetivo *URL* necessário para que o utilizador consiga solicitar a sua execução remota.
- `fetchFromServer(url: String): Future[String]`: realiza um *fetch* para o *URL* gerado pela função `buildUrl` e retorna a sua resposta.
- `createAndAppendDiv(htmlDiv: String): Unit`: utiliza o código *HTML* fornecido pelo `generateHtml` parar gerar as secções *Div* com os resultados dos comandos executados e associar ao *widget* de modo que o utilizador os consiga visualizar.
- `remoteCall(service: String, cmd: String, callback: String => Unit, ip: String, onError: String => Unit): Unit`: executa a função `fetchFrom`

`Server` e, com o resultado obtido, gera a respetiva secção *Div* e associa-a ao *widget* "Remote". Em caso de falha, retorna um erro a informar que não foi possível executar o comando solicitado.

5.3. Implementação do PR

Como mencionado anteriormente no Capítulo 3, o PR é um programa que foi desenvolvido para ilustrar e testar a funcionalidade da extensão do CAOS. Considerando a sua arquitetura, descrita na Secção 5.1, são apresentados abaixo os detalhes da sua implementação e inclui os elementos do *website*, análise do programa inserido, preparação dos pedidos a realizar e processamento das respostas recebidas. O código encontra-se disponível em `https://github.com/RaidenEi2020/Process-Runner`.

Elementos do *website*:

Relembrando a Figura 3.1, o PR é constituído por três elementos interativos:"Input program", *widget* onde é inserido o programa e cujo *parser* é descrita posteriormente, lista de exemplos de programas e *widget* "Remote", utilizado para visualizar os resultados da execução dos comandos solicitados. Estes dois últimos elementos encontram-se definidos no `CaosConfig.scala`, sendo as suas definições:

1. **Lista de exemplos**:

```
val examples = List(
    "ex1" -> "# comment\nserver: localhost:8080\npwd",
    ...
)
```

   A definição de cada programa exemplo consiste na associação do seu nome com o respetivo código.

2. ***Widget* "Remote"**:

```
val widgets = List(
    "Remote" -> viewRemote[Program](Remote.buildCommandList, Remote.generateHtml),
)
```

   Para que o *widget* "Remote" seja apresentado na interface do utilizador, é essencial incluir a sua definição na lista `widgets` (representada pela variável `widgets`), caracterizada por ser do tipo `viewRemote`, explicado na secção anterior, e suporta uma sintaxe do tipo `Program`. Para a sua geração, também é necessário passar duas funções como argumentos, definidas no ficheiro `Remote.scala`, que se encontra no

package `frontend`: **(1)** `buildCommandList(syntax:Map[String,List[String]]): List[(String, String)]`, que utiliza o resultado da análise do *parser* para preparar as solicitações que serão realizadas para o servidor, transformando-o numa lista genérica de pares `[(String, String)]`, e **(2)** `generateHtml(reply: String): String`, que processa cada uma das respostas recebidas através de *splits*, retornando o respetivo código *HTML* com o comando executado e o resultado da sua execução. Esta última função, após o seu processamento, é também capaz de obter o *id* do processo e o *token* do cliente que podem ser utilizados, por exemplo, em fases de *debugging* de um novo *widget*.

Parsing:

Um programa, quando inserido, é analisado pelo *parser* definido no `CaosConfig.scala`:

```
val parser: String => Program =
    clientPr.syntax.Parser.parseProgram
```

Este *parser*, recorrendo à utilização de combinadores de *parsing* fornecidos pela biblioteca `cats-parse`, analisa e converte o programa numa sintaxe do tipo `Program` que consiste num `Map` cuja chave é o servidor e o valor é a lista de comandos que serão executados. Por exemplo, se o utilizador inserir o programa:

```
server: localhost:8080
ls
pwd
```

o *parser* gera como resultado o `Map` $\{$`"localhost:8080"` $\mapsto$ $[$`"ls","pwd"`$]\}$.

5.4. Implementação de outro cliente - RebeCaos

O RebeCaos, como mencionado anteriormente, é uma *framework* que permite a análise de modelos na linguagem Rebeca, mas, como foi desenvolvida sobre o CAOS, apresenta os mesmos desafios: caso se pretenda adicionar novas análises, a sua implementação deve ser realizada em Scala ou JavaScript, sendo necessário um *browser web* que suporte JavaScript para as correr. A integração de novas análises no RebeCaos poderia ser útil, no entanto, para ilustrar a invocação de uma *framework* ou programa que possa ser utilizado através da execução de comandos recorreu-se a um exemplo mais simples: `wc`, comando que conta palavras e se encontra disponível em sistemas Unix. A Figura 5.3 ilustra o resultado da integração do RebeCaos com a extensão do CAOS.

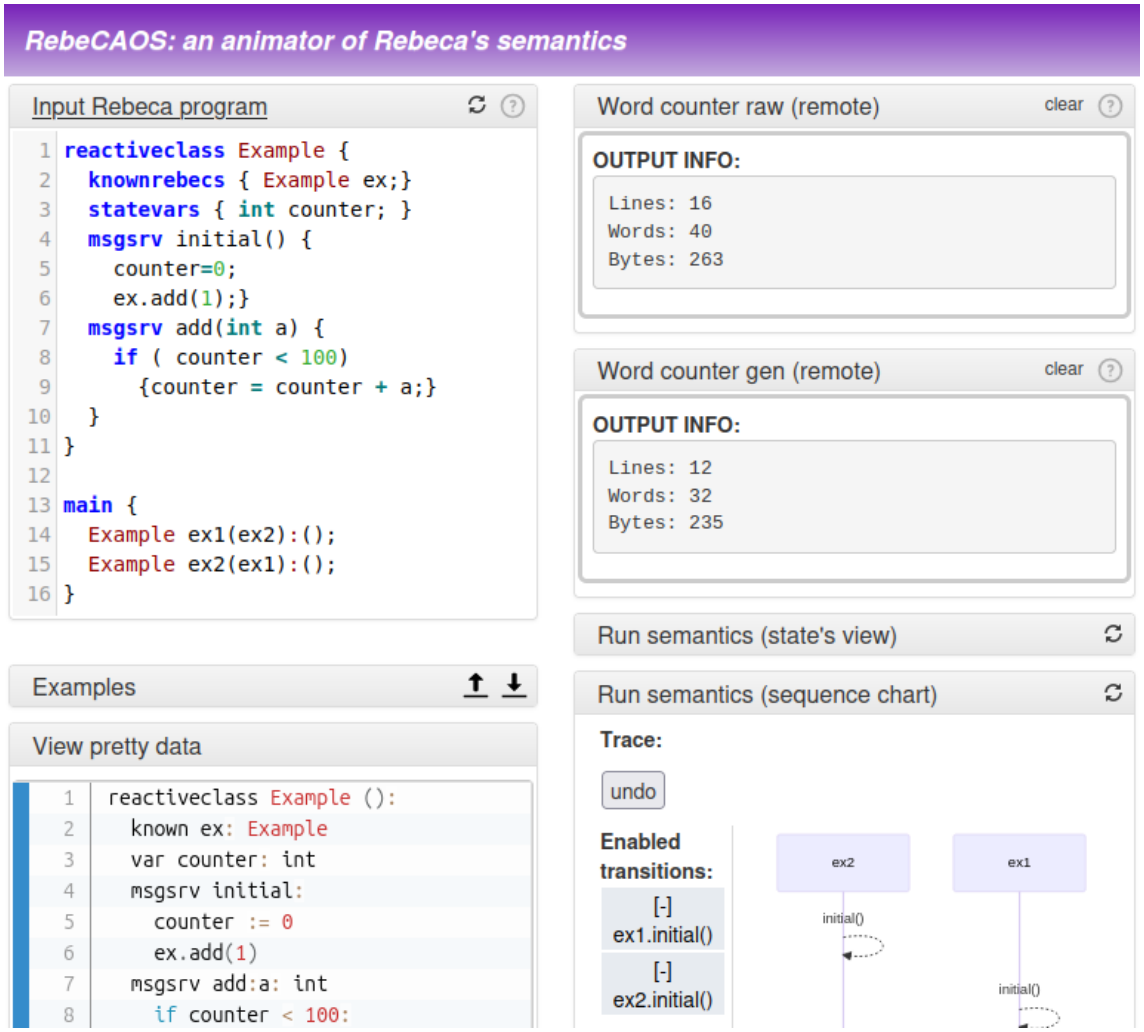


Figura 5.3: Captura de ecrã do RebeCaos usando a extensão do CAOS.

Como se pode observar, esta integração resultou na apresentação dois novos *widgets* na interface do utilizador: **(1)** "Word counter raw (remote)", que solicita a execução do comando `wc` para contar o número de palavras, linhas e *bytes* do programa e mostra o seu resultado, e **(2)** "Word counter gen (remote)", que funciona de forma semelhante ao anterior, mas aplicado à versão gerada pelo "View pretty data". Para a integração da extensão do CAOS com o RebeCaos e implementação destes dois novos *widgets*, foram realizados os passos descritos abaixo. O código encontra-se disponível em `https://github.com/RaidenEi2020/rebecaos`.

1. **Adicionar os *widgets* à estrutura do *website***: no ficheiro `Configurator.scala`, foram adicionadas as funções `viewRemoteRaw[Stx]`, que representa o "Word counter raw (remote), e `viewRemote[Stx]`, que representa o "Word counter gen (remote)". Como os *widgets* são semelhantes, foi implementada a função `viewRemoteAux[Stx]`,

utilizada para estabelecer a ligação entre eles de modo que a assinatura de ambos seja a `VisualizeRemote[Stx]`, descrita na passo seguinte.

`viewRemoteRaw[Stx]`:

```
def viewRemoteRaw[Stx](buildCmd: String => String, generateHtml: String => String,
    remember: Boolean = false, service: String = "localhost:8080"):
  WidgetInfo[Stx] = {
      viewRemoteAux(Right(buildCmd), None, generateHtml, remember, service)
  }
```

`viewRemote[Stx]`:

```
def viewRemote[Stx](buildCmd: (Stx, Stx => String) => String,
    prettyPrinter: Stx => String, generateHtml: String => String,
    remember: Boolean = false, service: String = "localhost:8080"):
  WidgetInfo[Stx] = {
      viewRemoteAux[Stx](Left(buildCmd), Some(prettyPrinter), generateHtml,
      remember, service)
  }
```

`viewRemoteAux[Stx]`:

```
private def viewRemoteAux[Stx](
    buildCmd: Either[(Stx, Stx => String) => String, String => String],
    prettyPrinter: Option[Stx => String], generateHtml: String => String,
    remember: Boolean = false, service: String): WidgetInfo[Stx] =
  VisualizeRemote[Stx](buildCmd, prettyPrinter, generateHtml, remember, service)
```

Para a definição das três funções, foram utilizados argumentos que são idênticos aos descritos no primeiro passo da Secção 5.2, como o `generateHtml: String => String` e o `remember: Boolean`. No entanto, também foram adicionados outros, sendo estes:

- `buildCmd: String => String`: função que gera o comando que será executado. Na `viewRemote[Stx]`, o seu tipo é `(Stx, Stx => String) => String`, dado que recebe a sintaxe resultante da análise do programa, mas necessita de uma função para a analisar e converter para na versão do *widget* "View Pretty Data", retornando-a na forma de `String`. Na `viewRemoteAux[Stx]`, o seu tipo é `Either[(Stx, Stx => String) => String, String => String]` para possibilitar a ligação entre os novos *widgets*.
- `prettyPrinter: Stx => String`: converte a sintaxe obtida da análise do programa para a versão do *widget* "View pretty data". Na `viewRemoteAux[Stx]`, o seu tipo é `Option[Stx => String]`, porque apenas um dos *widgets* implementados ("Word counter gen (remote)") utiliza esta função.

- `service: String`: variável com o *ip* do servidor. Caso não seja fornecido, é atribuído o valor `localhost:8080`.

2. **Definir a sua assinatura**: como os *widgets* se encontram interligados através da `viewRemoteAux[Stx]`, foi necessário definir apenas uma assinatura, `VisualizeRemote[Stx]`, no ficheiro `WidgetInfo.scala`. Todos os argumentos na sua definição são idênticos aos descritos no passo anterior.

```
case class VisualizeRemote[Stx](
    buildCmd: Either[(Stx, Stx => String) => String, String => String],
    prettyPrinter: Option[Stx => String], generateHtml: String => String,
    remember: Boolean, service: String)
  extends WidgetInfo[Stx]
```

3. **Atualizar a lógica do *website***: assim como na integração da extensão do CAOS com o PR, foi adicionado no ficheiro `Site.scala` o caso onde sempre que a assinatura é chamada, o construtor `RemoteVisualizeText[Stx]`, descrito no passo seguinte, é inicializado.

```
case VisualizeRemote(buildCmd, prettyPrinter, generateHtml, remember, service) =>
    new RemoteVisualizeText(buildCmd, prettyPrinter, getRaw, get, generateHtml,
        remember, service, w._1,out,doc)
```

Como se pode observar, para definir este caso foi necessária a utilização de argumentos idênticos ao descritos no primeiro passo, `getRaw`, `get`, `w._1`, `out` e `doc`, utilizados para obter o programa, sintaxe resultante da sua análise, nome do *widget*, caixa com os erros obtidos e documentação, respetivamente.

4. **Implementação do construtor de ambos os *widgets***: como ambos os *widgets* são semelhantes, apenas foi necessária a implementação de um construtor que se encontra no ficheiro `RemoteVisualizeText[Stx].scala` e cuja definição é:

```
class RemoteVisualizeText[Stx](
    buildCmd: Either[(Stx, Stx => String) => String, String => String],
    prettyPrinter: Option[Stx => String], raw: () => String, parsedRaw: () => Stx,
    generateHtml: String => String, remember: Boolean, service: String,
    name: String, errorBox: OutputArea, doc: Documentation)
  extends Widget[Unit](name, doc)
```

Como se pode observar no bloco de código apresentado, na definição deste construtor são utilizados argumentos idênticos aos descritos anteriormente, e para a geração dos novos *widgets*, elementos e respetivas interações foram implementadas funções idênticas às descritas na SecçãoSecção 5.2. A `buildUrl: String` foi

modificada de modo que suportasse a geração do *URL* correspondente de cada solicitação realizada pelos *widgets* “Word counter raw (remote)” e “Word counter gen (remote)”.

5. **Definir as funções do objeto** `Remote`: assim como na integração da extensão do CAOS com o PR, foi adicionado o ficheiro `Remote.scala` dentro do *package* `frontend`, com o respetivo objeto, para definir as funções que são utilizadas como argumentos para a `viewRemoteRaw[Stx]` e `viewRemote[Stx]`, sendo elas:

   - `buildCommand(syntax: St, prettyPrinter: St => String = _.toString): String`: gera os comandos aplicados à versão do *widget* “View pretty data”.
   - `buildCommandRaw(syntax: String): String`: gera os comandos aplicados ao programa inserido.
   - `generateHtml(reply: String): String`: idêntica à função `generateHtml` descrita na Secção 5.2.

6. **Adicionar os *widgets* à configuração do** `CaosConfig.scala`: após a realização dos passos anteriores, é necessário adicionar os novos *widgets* à lista de *widgets* do *website* e, para tal, são utilizadas as funções implementadas no `Configurator.scala` e `Remote.scala`.

```
val widgets = List(
    ...
    "Word counter raw (remote)" -> viewRemoteRaw(Remote.buildCommandRaw,
    Remote.generateHtml),
    "Word counter gen (remote)" -> viewRemote[St](Remote.buildCommand,
    s => Show(s.\_1), Remote.generateHtml),
    ...
  )
```

6. Conclusão e trabalho futuro

Este capítulo apresenta as principais conclusões obtidas da implementação da extensão do CAOS, servidor que a acompanha e desenvolvimento do PR (Secção 6.1), assim como sugestões de melhorias extensões que podem ser realizadas no futuro (Secção 6.2).

6.1. Conclusão

O CAOS é uma *framework* e metodologia que fornece ferramentas que permitem ao utilizador analisar um dado modelo, contudo a implementação de novas análises tem de ser realizada em Scala ou JavaScript. No entanto, em projetos como o RebeCaos poderia ser útil utilizar um simulador nativo do Rebeca via CAOS assim como portar para o CAOS muitas das análises realizadas remotamente pelo Reolive. Assim, nesta dissertação estendi o CAOS através do desenvolvimento de **(1)** um servidor, que permite executar comandos do sistema operativo remotamente, **(2)** construtor de *widgets*, para gerar um *widget* que apresenta o resultado dos comandos executados, e **(3)** PR, programa simples utilizado para ilustrar a funcionalidade da extensão do CAOS.

A implementação da sua extensão permitiu a execução de comandos e apresentar o seu resultado no *widget* gerado pelo novo construtor de *widgets*. Para além disso, a sua integração em implementações atuais que utilizam o CAOS, como o RebeCaos, torna possível a invocação de funcionalidades de outros programas, caso estes tenham suporte para linha de comandos de forma remota e interativa.

6.2. Trabalho futuro

Apesar da extensão do CAOS permitir a execução de comandos do sistema operativo de forma remota e invocação de programas utilizáveis por essa via, é possível identificar diversas melhorias que podem ser realizadas em versões futuras para a tornar mais eficiente, escalável e flexível. Algumas delas são:

- **Resultados parciais**: no seu estado atual, a extensão do CAOS apresenta o resultado da execução remota de comandos após a sua conclusão. Uma melhoria consistiria em fornecer ao utilizador melhor interatividade e a possibilidade de acompanhar processos longos em tempo real. Para tal, a interface poderia ser atualizada de forma periódica, onde o servidor enviava ao cliente respostas com o resultado

parcial e estado de execução, ou manual, através da implementação de mecanismos como um botão de atualização.

- **Otimização**: apesar do servidor desenvolvido suportar múltiplos utilizadores, em casos de elevada concorrência pode existir algumas latências. Uma possível abordagem que auxilia a mitigá-las seria a implementação de mecanismos de otimização, como a utilização de *cache*, presente em sistemas RESTful, e um ou mais balanceadores de carga com diversos servidores, melhorando a sua eficiência.

- **Implementação de testes**: para esta dissertação, foram realizados testes de demonstração para ilustrar a funcionalidade da extensão do CAOS. No entanto, poderia ser útil implementar testes automáticos (unitários, integração ou de carga) que permitam detetar limitações, falhas e auxiliem a garantir o funcionamento correto desta extensão.

- **Segurança**: como mencionado anteriormente, uma das rotas é `/admin-panel`, utilizada para fornecer aos administradores uma página que os auxilie na gestão do servidor, mas no seu estado atual, qualquer utilizador que tenha conhecimento desta rota tem acesso a essas ferramentas. Em versões futuras, para garantir integridade e confidencialidade, torna-se essencial a implementação de mecanismos de segurança como autenticação de utilizadores e controlo de acessos.

Para além das melhorias mencionadas, estender o sistema pode oferecer oportunidades para incorporar novas funcionalidades que melhoram a sua adaptabilidade e permitem a expansão gradual das suas capacidades. Possíveis extensões que podem ser realizadas são:

- **Customização do servidor**: o servidor desenvolvido permite a execução remota de comandos e a invocação de projetos que possam ser utilizados por esta via, mas este no seu estado atual não é customizável. Suportar a customização do servidor através de melhor documentação ou funções auxiliares tornaria-o mais reutilizável e flexivel.

- **Correr o código na Java Virtual Machine (JVM)**: atualmente, as análises que o CAOS fornece correm em Scala ou JavaScript, mas, em casos onde elevado desempenho é essencial, pode ser vantajoso executar o mesmo código Scala na JVM em vez ser compilado para JavaScript. Esta extensão tornaria o CAOS mais

eficiente e auxiliaria na integração de novas funcionalidades implementadas em Java.

- **Rede com múltiplos servidores**: a extensão do CAOS realizada nesta dissertação permite a execução de comandos remotamente em um ou mais servidores individuais. Uma possível abordagem que melhoraria o seu serviço seria o desenvolvimento de uma rede distribuída com múltiplos servidores capazes de interagir entre si. Esta rede tornaria o serviço mais eficiente, escalável, tolerável a falhas e auxiliaria na sua customização.

- **Interface que suporte diferentes tipos de conexão ao servidor**: apesar do servidor não ser completamente RESTful como mencionado anteriormente, a sua comunicação cliente-servidor é simples. Estender a sua comunicação e a interface do utilizador de modo que suporte diferentes tipos de conexões e protocolos (por exemplo *SSH*) não só auxiliaria na integração de funcionalidades de aplicações externas como também possibilitaria ao cliente utilizar a comunicação mais adequada às suas necessidades.

Referências Bibliográficas

[1] J. Proença and L. Edixhoven, “The caos framework for scala: computer-aided design of sos,” *Science of Computer Programming*, vol. 103222, p. 103222, 2024. [Online]. Available: https://doi.org/10.1016/j.scico.2024.103222 [Cited on pages 1 and 4.]

[2] J. Proença and L. Edixhoven, “Caos: A reusable scala web animator of operational semantics,” in *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Portugal, June 19-23, 2023, Proceedings*, ser. Lecture Notes in Computer Science, S.-S. Jongmans and A. Lopes, Eds., vol. 13908. Springer, 2023, pp. 163–171. [Online]. Available: https://doi.org/10.1007/978-3-031-35361-1_9 [Cited on pages 1 and 4.]

[3] R. Cruz and J. Proença, “Reolive: Analysing connectors in your browser,” in *Software Technologies: Applications and Foundations - STAF 2018 Collocated Workshops, Toulouse, France, June 25-29, 2018, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Mazzara, I. Ober, and G. Salaün, Eds., vol. 11176. Springer, 2018, pp. 336–350. [Online]. Available: https://doi.org/10.1007/978-3-030-04771-9_25 [Cited on pages 1 and 4.]

[4] M. ter Beek and J. Proença, “Rebecaos,” in *Proceedings of COORDINATION 2025, Lille, France, 17th of June 2025*, ser. Lecture Notes in Computer Science, C. D. Giusto and A. Ravara, Eds., vol. 15731. Springer, 2025, pp. 219–229. [Online]. Available: https://doi.org/10.1007/978-3-031-95589-1_11 [Cited on pages 1 and 3.]

[5] M. Sirjani and M. M. Jaghoori, “Ten years of analyzing actors: Rebeca experience,” in *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, ser. Lecture Notes in Computer Science, G. Agha, O. Danvy, and J. Meseguer, Eds., vol. 7000. Springer, 2011, pp. 20–56. [Online]. Available: https://doi.org/10.1007/978-3-642-24933-4_3 [Cited on page 1.]

[6] M. ter Beek and J. Proença, “Animating rebeca,” in *”Rebeca for Actor Analysis in Action: Essays Dedicated to Marjan Sirjani on the Occasion of Her 60th*

Birthday", ser. Lecture Notes in Computer Science, E. Lee, M. R. Mousavi, and C. Talcott, Eds., vol. 15560. Springer, 2025, pp. 182–194. [Online]. Available: https://doi.org/10.1007/978-3-031-85134-6_8 [Cited on page 1.]

[7] E. Khamespanah, M. Sirjani, and R. Khosravi, "Afra: An eclipse-based tool with extensible architecture for modeling and model checking of rebeca family models," in *Fundamentals of Software Engineering (FSEN 2023)*, ser. Lecture Notes in Computer Science, H. Hojjat and E. Ábrahám, Eds., vol. 14155. Springer, 2023, pp. 72–87. [Cited on pages 1 and 2.]

[8] S. Goncharov, R. Neves, and J. Proença, "Implementing hybrid semantics: From functional to imperative," in *Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau S.A.R., China, October 31 - November 30, 2020, Proceedings*, ser. Lecture Notes in Computer Science, K. I. Pun, A. da Silva Simão, and V. Stolz, Eds., vol. 12545. Springer, 2020. [Cited on pages 1 and 4.]

[9] P. Mendes, R. Correia, R. Neves, and J. Proença, "Formal simulation and visualisation of hybrid programs," in *Proceedings Sixth International Workshop on Formal Methods for Autonomous Systems, FMAS@iFM 2024, Manchester, UK, 11th-13th of November 2024*, ser. EPTCS, M. Luckcuck and M. Xu, Eds., vol. 411, 2024, pp. 20–37. [Online]. Available: https://doi.org/10.4204/EPTCS.411.2 [Cited on pages 1 and 4.]

[10] J. Proença and A. Madeira, "Taming hierarchical connectors," in *Fundamentals of Software Engineering - 8th International Conference, FSEN 2019, Tehran, Iran, May 1-3, 2019, Revised Selected Papers*, ser. Lecture Notes in Computer Science, H. Hojjat and M. Massink, Eds., vol. 11761. Springer, 2019, pp. 186–193. [Online]. Available: https://doi.org/10.1007/978-3-030-31517-7_13 [Cited on page 4.]

[11] M. H. ter Beek, R. Hennicker, and J. Proença, "Realisability of global models of interaction," in *Theoretical Aspects of Computing - ICTAC 2023 - 20th International Colloquium, Lima, Peru, December 4-8, 2023, Proceedings*, ser. Lecture Notes in Computer Science, E. Ábrahám, C. Dubslaff, and S. L. T. Tarifa, Eds., vol. 14446. Springer, 2023, pp. 236–255. [Online]. Available: https://doi.org/10.1007/978-3-031-47963-2_15 [Cited on page 4.]

[12] G. Cledou, J. Proença, B. H. C. Sputh, and E. Verhulst, "Hubs for virtuosonext: Online verification of real-time coordinators," *Science of Computer Programming*,

vol. 203, p. 102566, 2021. [Online]. Available: https://doi.org/10.1016/j.scico.2020.102566 [Cited on page 4.]

[13] M. H. ter Beek, G. Cledou, R. Hennicker, and J. Proença, "Can we communicate? using dynamic logic to verify team automata," in *Proceedings of the 25th International Symposium on Formal Methods (FM 2023)*, ser. Lecture Notes in Computer Science, M. Chechik, J.-P. Katoen, and M. Leucker, Eds., vol. 14000. Springer, 2023. [Cited on page 4.]

[14] G. Cledou, J. Proença, and L. S. Barbosa, "Composing families of timed automata," in *Fundamentals of Software Engineering - 7th International Conference, FSEN 2017, Tehran, Iran, April 26-28, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Dastani and M. Sirjani, Eds., vol. 10522. Springer, 2017, pp. 51–66. [Online]. Available: https://doi.org/10.1007/978-3-319-68972-2_4 [Cited on page 4.]

[15] GeeksforGeeks. (2025, Jun.) Rest api introduction. [Online]. Available: https://www.geeksforgeeks.org/rest-api-introduction [Cited on page 5.]

[16] RestfulAPI.net. (2025, Apr.) Rest api tutorial: What is rest? [Online]. Available: https://restfulapi.net [Cited on page 5.]

[17] S. Gupta. (2022, Dec.) http4s for functional http. [Online]. Available: https://blog.nashtechglobal.com/http4s-for-functional-http/ [Cited on page 8.]

[18] Baeldung. (2025, Apr.) Introduction to http4s. [Online]. Available: https://www.baeldung.com/scala/http4s-intro [Cited on page 8.]