Como ofertar aplicações *web* Python como um provedor de serviço Shibboleth usando microsserviços

Felipe dos Passos Cardoso[2,3], Allex Magno Andrade[2,3], Emerson Ribeiro de Mello[2], Carolina Howard Felicissimo[3], Michelle Silva Wangham[1,3]

[1]Universidade do Vale do Itajaí (UNIVALI) – São José, SC – Brasil

[2]Instituto Federal de Santa Catarina (IFSC) – São José, SC – Brasil

[3]Rede Nacional de Ensino e Pesquisa (RNP)

{felipe.p19,allex.m}@aluno.ifsc.edu.br, mello@ifsc.edu.br,

carolina.felicissimo@rnp.br, wangham@univali.br

Abstract. *The Shibboleth framework provides a module for the Apache HTTP server to deliver SAML assertions to your applications transparently. Python web applications are commonly offered on the Internet through two servers: one application, which implements WSGI, and an HTTP, which acts as a proxy. In the microservice architecture, it is desired that the servers are distributed in such a way that there is only one service running per container. This work presents a form of container composition to offer a Python web application as a Shibboleth Service Provider using the microservices architecture.*

Resumo. *O framework Shibboleth fornece um módulo para o servidor HTTP Apache que permite que este entregue, de forma transparente, asserções SAML para suas aplicações. Aplicações web Python são comumente disponibilizadas na Internet por meio de dois servidores: um de aplicação, que implementa WSGI; e um HTTP, que atua como proxy. Na arquitetura de microsserviços, é desejado que os servidores, necessários para execução de aplicação, sejam distribuídos de forma que se tenha somente um serviço em execução por contêiner. Este trabalho apresenta uma forma de composição de contêineres para ofertar uma aplicação web Python como um provedor de serviço Shibboleth dentro do contexto da arquitetura de microsserviços.*

1. Introdução

A gestão de identidade (GId)[1] pode ser entendida como um conjunto de políticas, processos e tecnologias para prover identificação, autenticação, autorização e auditoria [ITU 2009]. Os modelos de GId guiam a concepção de infraestruturas de autenticação e de autorização, mas não indicam quais protocolos ou tecnologias devem ser adotados. Dentre os modelos, o de gestão de identidade federada oferece alguns benefícios que melhoram a usabilidade para os usuários de aplicações *web*, como o fato de o usuário ter uma única credencial para acessar diversos provedores de serviço e só passar pelo processo de autenticação uma única vez durante a sessão [Wangham e et al. 2018].

[1]Também conhecida como gestão de identidade de acesso (*Identity and Access Management* - IAM).

Sistemas para GId utilizam de padrões para assegurar a qualidade das informações trocadas e garantir a interoperabilidade entre seus componentes. A especificação SAML 2.0 se destaca nesse contexto padronizando regras e uma estrutura para troca de asserções com informações de segurança entre as entidades envolvidas [OASIS 2008]. Contudo, fazer uso de asserções SAML em mecanismos de autenticação e autorização de aplicações *web* não é uma tarefa trivial. Mesmo com uso de bibliotecas que implementem o padrão, os processos de estudo e adaptação consomem tempo extra para seu desenvolvimento. O *framework* Shibboleth provê um *middleware* para provedores de serviço (SPs) possibilitando que desenvolvedores utilizem SAML em suas aplicações abstraindo grande parte dessas dificuldades. O *middleware* está disponível como um módulo para o servidor HTTP Apache[2] e permite que aplicações consumam asserções SAML de maneira transparente através de variáveis de sessão HTTP [Chaves e de Mello 2015].

Aplicações *web* Python podem ser servidas pelo Apache, por meio do módulo WSGI, e assim obter acesso às variáveis de ambiente criadas pelo SP Shibboleth ao consumir uma asserção SAML. Porém, a configuração mais comum para disponibilizá-las na Internet é a utilização de dois servidores em conjunto: um servidor que implemente WSGI nativamente, responsável por hospedar o código-fonte da aplicação; e um HTTP que serve como *proxy*, com a função de tratar solicitações da *web*. Em uma arquitetura de microsserviços, as aplicações são organizadas como uma coleção de serviços independentes com baixo acoplamento entre si [Fowler e Lewis 2014]. Com isso, na configuração descrita anteriormente, é desejável que os servidores sejam implantados de maneira isolada com apenas um serviço por contêiner.

Este trabalho tem por objetivo descrever uma solução que usa uma composição de contêineres e a arquitetura de microsserviços para permitir que aplicações *web* Python possam consumir asserções SAML, através do *framework* Shibboleth, sem a necessidade de hospedá-las no servidor HTTP Apache. A solução possibilita que a aplicação seja desenvolvida em diferentes *frameworks* Python, tais como Django, Flask e TurboGears.

A Seção 2 deste artigo apresenta os conceitos e tecnologias envolvidos na concepção da solução proposta. A Seção 3 descreve a solução baseada em contêineres para prover autenticação SAML em aplicações *web* Python. Na Seção 4, o uso da solução proposta no estudo de caso do LIneA é analisado. Informações sobre a documentação da solução e como será a demonstração durante o SBSeg são descritas na Seção 5. Por fim, as conclusões e a indicação de trabalhos futuros são apresentadas na Seção 6.

2. Conceitos e Tecnologias

Security Assertion Markup Language (SAML) é uma especificação da *Organization for the Advancement of Structured Information Standards* (OASIS) que define um conjunto de regras e uma estrutura baseada em XML para troca de informações de segurança (asserções) entre entidades parceiras [OASIS 2008]. Com esse padrão, é possível implementar os modelos de GId centralizado e federado. Dentre os perfis oferecidos, destacam-se o *Web Browser Single Sign-On* (SSO), que permite a autenticação única do usuário para acessar diferentes serviços, e o *Single Log-Out* (SLO), que possibilita ao usuário encerrar a sessão em todos os serviços em uma única vez [Wangham et al. 2010]. Um sistema de gestão de identidade envolve os seguintes componentes:

[2] `https://httpd.apache.org/`

- **Usuário**: entidade que deseja acessar e consumir recursos fornecidos pelos Provedores de Serviço;
- **Identidade:** descreve as entidades por meio de atributos, fornecendo sua identificação e características;
- **Provedor de Serviço (SP):** oferece recursos a usuários autorizados, após verificar a autenticidade de sua identidade e comprovar que esta carrega todos os atributos necessários para o acesso;
- **Provedor de Identidades (IdP):** responsável por gerenciar identidades de entidades e pelo processo de autenticação;

No modelo de identidades federadas, relações de confiança são estabelecidas entre IdPs e SPs para possibilitar a troca de informações relacionadas a identidade e o compartilhamento de serviços. Os acordos de confiança existentes garantem que usuários autenticados em seus IdPs de origem possam acessar recursos protegidos providos por SPs de outros domínios da federação [Wangham et al. 2010].

Existem diversas soluções que implementam federações baseadas na especificação SAML, como é o exemplo do Shibboleth desenvolvido pela Internet2 em 2000. O *framework* Shibboleth contêm dois componentes principais: um software que implementa o provedor de identidade (IdP Shibboleth) e um *middleware* para implementação de provedores de serviços (SP Shibboleth). O IdP Shibboleth foi desenvolvido na linguagem Java e fornece opções para políticas de autenticação e liberação de atributos, operando como um cliente da base de usuários, normalmente, o *Active Diretory* ou *OpenLdap*. Já o SP Shibboleth é um módulo para o servidor HTTP Apache (`mod_shib`), executado pelo *shib daemon* (`shibd`), que é independente de linguagem. Ou seja, serviços desenvolvidos em qualquer linguagem podem fazer uso do *framework* sem prejuízos [Scavo et al. 2005].

O `mod_shib` permite que os recursos das aplicações servidas pelo Apache sejam protegidos e liberados através de uma sessão válida, obtida por meio das asserções SAML. Os SPs desenvolvidos em PHP possuem suporte nativo do Apache e podem consumir essas asserções diretamente. SPs implementados em outras linguagens necessitam ser integrados através de módulos. Assim, SPs em Java utilizam o AJP (`mod_proxy_ajp`) e os desenvolvidos em Python, o WSGI (`mod_wsgi`) [Kubica 2009].

Web Server Gateway Interface (WSGI) é uma especificação de interface para comunicação entre servidores *web* e aplicações Python. A interface possui dois lados, o da aplicação e o do servidor. A implementação no lado da aplicação precisa fornecer um objeto, definido por meio de um arquivo (*script*) denominado `app.wsgi` ou `wsgi.py`, que será chamado pelo lado do servidor ao receber requisições [Eby 2010]. O Apache possui o `mod_wsgi` que provê o lado servidor da interface, porém, não é muito utilizado para servir aplicações *web* Python. A estrutura mais usada na Internet possui dois componentes: (1) *frontend* que contém um servidor HTTP (Apache ou NGINX[3]) como *proxy* para tratar as requisições da *web* e redirecioná-las ao servidor de aplicação; e (2) *backend* com um servidor de aplicação que implementa o padrão WSGI, como o uWSGI *Server*.

O uWSGI *Server* é um projeto, mantido pela Unbit[4], com objetivo de construir uma plataforma completa para hospedagem de serviços com uma arquitetura facilmente estendida através de *plugins*. Apesar de implementar diversos padrões, o WSGI foi o

[3] `http://nginx.org/en/`
[4] `https://unbit.it/`

primeiro a ser adotado e seu nome escolhido em forma de homenagem. O servidor implementa nativamente o *uwsgi* (minúsculo), protocolo binário para transferência de dados que utiliza comunicação via *Trasmission Control Protocol* (TCP) entre as partes [Unbit 2020]. Este protocolo também é implementado pelo Apache através do módulo *proxy uwsgi* (`mod_proxy_uwsgi`).

Para aplicações Python *web* desenvolvidas com Django[5], a *Brown University*[6] implementou um *middleware* para integração com o SP Shibboleth. O `django-shibboleth-remoteuser`[7] estende os mecanismos de autenticação nativos do Django e seu funcionamento permite a criação de objetos do modelo de usuário utilizando os atributos extraídos de uma asserção SAML. Com isso, aplicações já desenvolvidas podem incluir um novo padrão de autenticação sem a necessidade de alterar as regras de negócio existentes ou implementar novas.

A arquitetura de microsserviços é utilizada para projetar e desenvolver aplicações como um conjunto de pequenos serviços que são implementados de maneira independentes. Além disso, os serviços possuem um baixo acoplamento entre si e se comunicam através de protocolos leves, o que possibilita a escalabilidade dos sistemas e também que sejam desenvolvidos em diferentes linguagens de programação [Fowler e Lewis 2014].

A conteinerização é um método de virtualização que consiste em empacotar a aplicação e suas dependências de forma a proporcionar o isolamento do conjunto. Com isso, essa técnica viabiliza implantações que seguem a abordagem de microsserviços e permite a virtualização em massa de sistemas computacionais, procedimento muito utilizado em *Platform-as-a-Service* (PaaS) e *Infrastructure-as-a-Service* (IaaS). Em 2013, a empresa *Docker* disponibilizou uma ferramenta de mesmo nome que impulsionou a implantação de aplicações conteinerizadas [de Oliveira e Miers 2019].

3. Solução Proposta

A solução proposta tem como objetivo disponibilizar uma composição de contêineres personalizada para facilitar o desenvolvimento de aplicações *web* Python como SPs Shibboleth. Com isso, desenvolvedores não precisam se ocupar estudando bibliotecas nem mesmo migrar suas aplicações para o servidor HTTP Apache com o intuito de incluir a autenticação SAML. Ficam apenas com a tarefa de utilizar as variáveis de ambiente, extraídas de uma asserção SAML pelo SP Shibboleth, em suas regras de negócio ou incluir *middlewares* que as aproveitam, como o *django-shibboleth-remoteuser*.

Seguindo a arquitetura de microsserviços, a composição é montada com a finalidade de manter o isolamento das aplicações separando os serviços em dois contêineres que são orquestrados através da ferramenta *Docker Compose*[8]. O primeiro contêiner, chamado de `Proxy`, é construído com uma imagem predefinida contendo o servidor HTTP Apache e o *middleware* SP Shibboleth. Já o segundo, denominado `Backend`, tem a imagem estruturada com uma aplicação *web* Python de exemplo (em Django, TurborGears ou Flask) e o servidor de aplicação uWSGI.

Os contêineres são executados na rede interna do *Docker* e se comunicam através

[5] `https://www.djangoproject.com/`
[6] `https://www.brown.edu/`
[7] `https://github.com/Brown-University-Library/django-shibboleth-remoteuser`
[8] `https://docs.docker.com/compose/`

de requisições entre os servidores utilizando *uwsgi*, um protocolo binário para transferência de dados. Além disso, o contêiner `Backend` está isolado da rede externa e recebe as solicitações passando pelo servidor Apache do contêiner `Proxy` que contém o `mod_shib` instalado.

A personalização é feita por meio da execução de um *script shell* (`create-compose.sh`) que irá solicitar algumas informações do ambiente e como resultado têm-se: o arquivo de descrição da composição (`docker-compose.yml`); os volumes e arquivos de configurações dos softwares utilizados; e a infraestrutura com os contêineres (ver Figura 1), a saber:

- `Proxy` - Contêiner com servidor HTTP Apache e o *middleware* SP Shibboleth
- `Backend` - Contêiner com servidor de aplicação uWSGI e uma aplicação *web* Python.

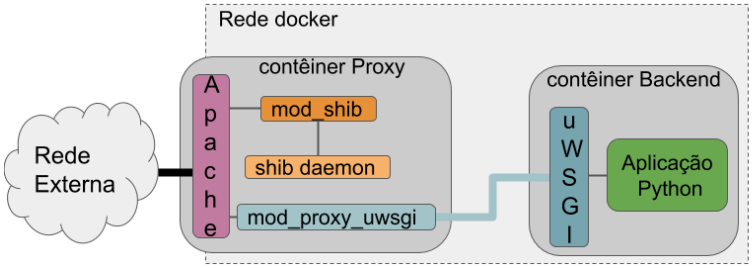

Figura 1. Composição da solução proposta

3.1. Contêiner Proxy

A imagem *Docker* utilizada para o contêiner `Proxy` tem como base o sistema operacional *Debian*, onde são executados o servidor HTTP Apache e o SP Shibboleth (`mod_shib` e *shib daemon*). Ainda são habilitados os módulos *proxy uwsgi* (`mod_proxy_uwsgi`) e TLS (`mod_ssl`).

O servidor Apache funciona como *proxy* para as requisições dos usuários externas à composição. Ou seja, é responsável por tratar as solicitações da Internet e encaminhá-las ao destino dentro da rede *Docker*, de acordo com as rotas definidas em seu arquivo de configuração. O Apache encaminha as requisições com destino ao `Backend` transferindo os dados através do protocolo *uwsgi* que é implementado pelo `mod_proxy_uwsgi`. O `mod_ssl` habilitado permite a comunicação criptografada entre o navegador *web* do usuário e o servidor. Esse contêiner pode ser utilizado para hospedar códigos *frontend* do serviço, porém isso foge do escopo desse trabalho.

O SP Shibboleth instalado no contêiner `Proxy` possui dois componentes: `mod_shib` e *shib daemon* (`shibd`). O `mod_shib` realiza a ligação entre o servidor Apache e o `shibd`. Dessa forma, é possível configurar o módulo para proteger *enpoints* das aplicações servidas pelo Apache, inclusive as servidas por redirecionamento, requisitando uma autenticação SAML. Ou seja, o recurso protegido que for solicitado será entregue somente se existir uma sessão válida, criada por uma asserção SAML. Caso o usuário não esteja autenticado ao tentar acessar o recurso, o módulo redireciona a requisição para o componente `shibd` que inicia o fluxo[9] de autenticação SAML. Após a autenticação, o

[9]Detalhes sobre o fluxo estão fora do escopo deste trabalho e por isso não serão tratados.

`shibd` recebe a asserção com os atributos do usuário e informações da sessão, define as variáveis de ambiente e redireciona o navegador do usuário para o recurso solicitado.

Na solução proposta, como os recursos a serem protegidos são da aplicação *web* Python disponibilizada no contêiner `Backend`, as variáveis de ambientes mencionadas anteriormente são transferidas via protocolo *uwsgi* para a aplicação. Isso permite que as informações extraídas da asserção SAML, consumida pelo SP Shibboleth, sejam usadas pelos desenvolvedores em suas regras de negócio para o controle de acesso do usuário.

3.2. Contêiner Backend

O contêiner `Backend` é estruturado com uma base Python (v.3.6) e o servidor uWSGI instalado. A solução fornece aplicações *web* Python de exemplo desenvolvidas com *frameworks* conhecidos, como Django, TurboGears[10] ou Flask[11]. Por isso, o contêiner não é executado a partir de uma imagem *Docker* predefinida e sim construída como última etapa da personalização, a partir da escolha do usuário na execução do *script*.

O uWSGI instalado no `Backend` está apto para servir aplicações que implementem o padrão WSGI. Isso é possível inserindo em seu arquivo de configuração (`uwsgi-server.ini`) a localização do *script* responsável por instanciar o objeto da aplicação que será invocado quando uma requisição for recebida. O servidor também está configurado para dispor a aplicação por meio do protocolo *uwsgi* utilizado para receber requisições enviadas pelo Apache do contêiner `Proxy`.

As variáveis de ambiente, criadas pelo SP Shibboleth no contêiner `Proxy`, transferidas para o *endpoint* protegido da aplicação, podem ser utilizadas com a garantia de terem sido obtidas em uma asserção SAML válida, depois de uma autenticação bem sucedida. Assim, ao utilizar os atributos disponíveis nessas variáveis, o desenvolvedor pode aplicar as regras de negócio daquela aplicação, como gerar uma sessão para o usuário autenticado, definir um *token* de acesso, criar um novo usuário entre outras.

A composição de contêineres proposta possui ainda as seguintes funcionalidades:

- **SLO** - permite o *logout* único em todas os serviços que o usuário possui uma sessão SAML ativa;
- **SP R&S** - adequado as exigências do *Research and Scholarship Entity Category*[12];
- **CAFe Expresso**[13] - habilitado para utilizar o serviço de descoberta (WAYF) da CAFe Expresso ao iniciar o processo de autenticação SAML.

4. Caso de Uso: LIneA

Um caso de uso para a solução proposta foi a inclusão da autenticação SAML em uma aplicação *web* Python do LIneA[14]. Nesse caso, a aplicação já possuía uma autenticação local e sua infraestrutura era composta pelos seguintes contêineres: (1) `Frontend` com uma aplicação JavaScript usando *React*[15] para interface com o usuário e o servidor NGINX.

[10] `https://turbogears.org/`

[11] `https://palletsprojects.com/p/flask/`

[12] Definida pela REFEDS (the Research and Education FEDerations group) esta categoria delimita a troca de informações de identidade em um conjunto de atributos a serem liberados pelos IdPs aos SPs

[13] Federação SAML para experimentação disponível em `https://gidlab.rnp.br/?page_id=916`

[14] Laboratório Interinstitucional de e-Astronomia - `https://www.linea.gov.br`

[15] `https://pt-br.reactjs.org/`

Esse contêiner também executa a função de *proxy* para as requisições com destino à aplicação; (2) `Backend` com aplicação Python desenvolvida em Django e o servidor Gunicorn[16].

Com a utilização da composição de contêineres descrita na solução proposta, foi possível manter as funcionalidades (dentre elas a autenticação local) e regras de negócio da aplicação realizando as alterações descritas a seguir:

1. Instalação do *middleware* `django-remote-user` na aplicação Django do contêiner `Backend`.
2. Inclusão do contêiner `Proxy` na composição.
3. Substituição do servidor Gunicorn pelo uWSGI no contêiner `Backend`. Essa alteração foi realizada pelo fato do Gunicorn não implementar o protocolo *uwsgi*, necessário para comunicação entre contêiner `Proxy` e o contêiner `Backend`.
4. Inserção de um botão para autenticação SAML (Login Institucional) na interface da aplicação do contêiner `Frontend`.
5. Configurações das rotas e protocolos de redirecionamento usados no servidor NGINX do contêiner `Frontend`.

As alterações resultaram numa nova composição com três contêineres como pode ser observada na Figura 2. O contêiner `Proxy` foi adicionado à infraestrutura inicial para incluir a autenticação SAML na aplicação. O SP Shibboleth trata as asserções SAML e cria as variáveis de ambiente no Apache que as redireciona, via protocolo *uwsgi*, para o *endpoint* protegido do middleware *django-shibboleth-remoteuser* responsável por criar a sessão na aplicação Django. Após isso, enquanto a sessão for válida, todas as requisições do usuário externas à composição com destino aos demais *endpoints* da aplicação são redirecionados pelo NGINX via protocolo *uwsgi*.

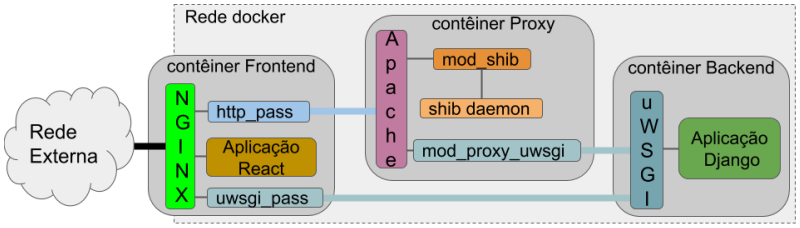

Figura 2. Composição do estudo de caso LIneA

5. Documentação e Demonstração

O *script shell* para personalização da composição pode ser obtido através do repositório do GIdLab[17], assim como documentos, requisitos e arquivos necessários para implantação. As imagens *Dockers* usadas na solução podem ser encontradas no *dockerhub*[18] junto com as informações necessárias para utilização.

Para demonstração da solução planejada para o Salão de Ferramentas, o *script* será executado para incluir a autenticação SAML em uma aplicação *Web* Python usando o *framework* Django e o *middleware* `django-shibboleth-remoteuser`. Um teste de autenticação para acesso à aplicação também será apresentado.

[16] `https://gunicorn.org/`
[17] Disponível em `https://git.rnp.br/gidlab/proxy-shibboleth`
[18] Disponível em `https://hub.docker.com/u/gidlab`

6. Conclusão

A inclusão da autenticação SAML em aplicações *Python* não é um processo fácil devido a necessidade do uso de bibliotecas que implementem o padrão. A solução proposta aproveita-se do *framework* Shibboleth para incluir essa funcionalidade e facilitar o processo de desenvolvimento de provedores de serviço SAML utilizando uma arquitetura de microsserviços. O estudo de caso apresentado na Seção 4 demonstra que é possível utilizar a solução até mesmo em aplicações existentes sem a necessidade de alterações das regras de negócio. Como trabalhos futuros, pretende-se adicionar suporte a aplicações Java e desenvolver uma interface *Web* para personalização da composição de contêineres.

7. Agradecimentos

Este trabalho foi desenvolvido no GIdLab[19], um serviço para experimentação em GId mantido pela Rede Nacional de Ensino e Pesquisas (RNP). Os autores agradecem ainda aos colaboradores do LIneA.

Referências

Chaves, S. A. e de Mello, E. R. (2015). Adoção de modelo controle de acesso baseado em atributos em sistema de votação online para ofertá-lo como um serviço tic federado. In *Anais do*. 15 Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg), 2015, SBC.

de Oliveira, K. d. S. V. e Miers, C. (2019). Uma análise de segurança no uso de contêineres do tipo docker em nuvens iaas openstack. In *Anais da XIX Escola Regional de Alto Desempenho da Região Sul*, Porto Alegre, RS, Brasil. SBC.

Eby, P. J. (2010). Python web server gateway interface v1.0.1. `https://www.python.org/dev/peps/pep-3333/`. Acessado em 26 jul. 2020.

Fowler, M. e Lewis, J. (2014). Microservices a definition of this new architectural term. *`http://martinfowler.com/articles/microservices.html`*, page 22. Acessado em 25 jul. 2020.

ITU (2009). Ngn identity management framework. Recommendation Y.2720.

Kubica, M. (2009). Howto use python in the web. `https://docs.python.org/3.0/howto/webservers.html`. Acessado em 24 jul. 2020.

OASIS (2008). Security assertion markup language(saml) v2.0 technical overview. Technical report.

Scavo, T., Cantor, S., e Dors, N. (2005). Shibboleth architeture: Technical overview. Technical report.

Unbit (2020). The uwsgi project. `https://uwsgi-docs.readthedocs.io/en/latest/`. Acessado em 26 jul. 2020.

Wangham, M. S., de Mello, E. R., da Silva Böger, D., Guerios, M., e da Silva Fraga, J. (2010). Gerenciamento de identidades federadas. In *Minicurso – SBSeg 2010*.

Wangham, M. S. e et al. (2018). O futuro da gestão de identidades digitais. In *Anais Estendidos do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 146–166, Porto Alegre, RS, Brasil. SBC.

[19] `https://gidlab.rnp.br`