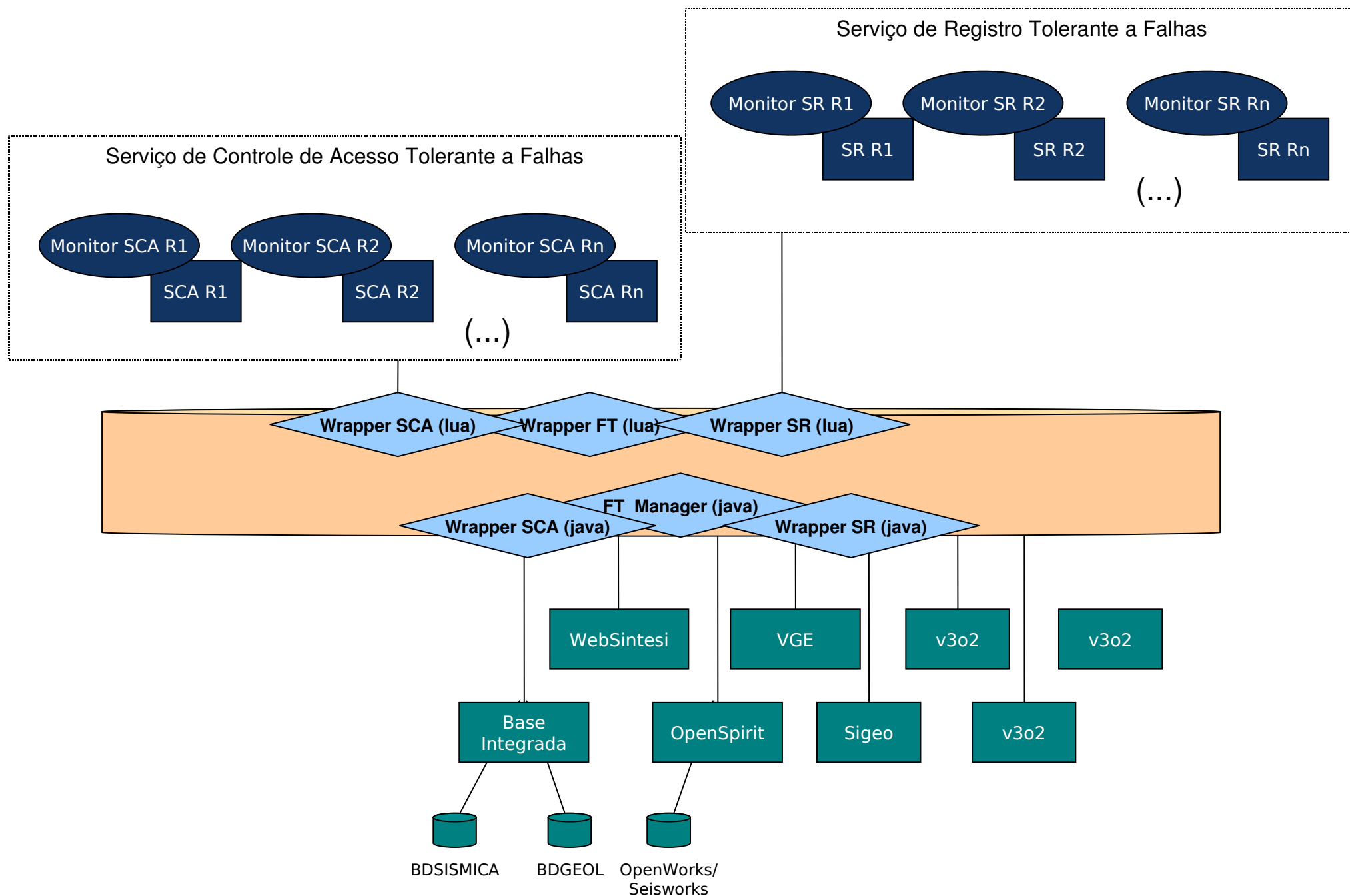
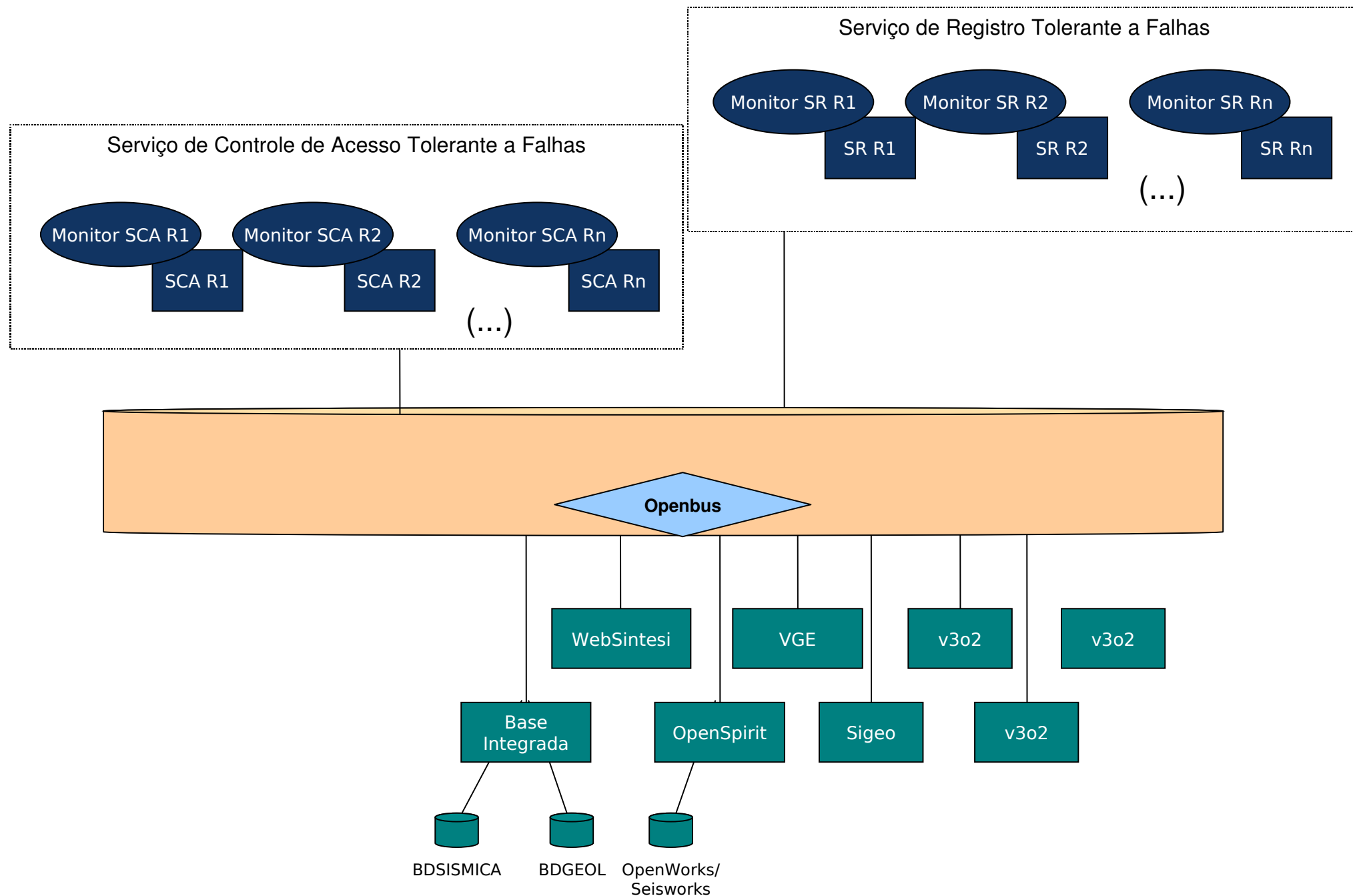


OpenBus - TF

Novembro/2009

Maíra Gatti



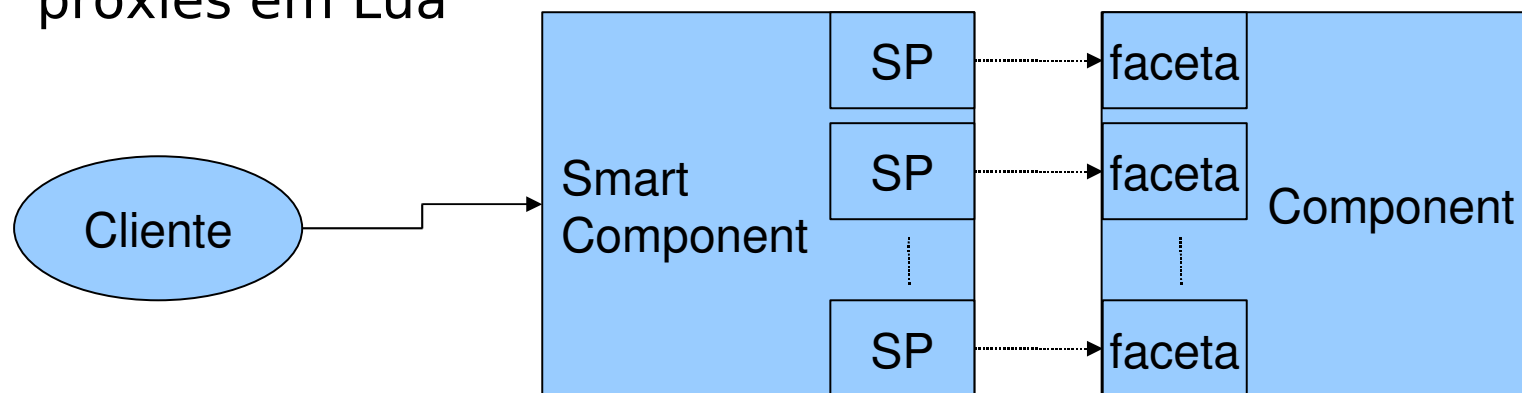


- Replicação Híbrida
 - Atualização do estado por demanda
 - Réplica líder escolhida por próxima réplica disponível em uma fila
- Lua: SmartsProxies e SmartComponent
 - Redireciona requisição para outra réplica em caso de falha
- Java e C++: Portable Interceptors
 - Trata exceção no `receive_exception`
 - Redireciona requisição para outra réplica em caso de falha via `LocationForward`
- Pode existir mais de uma réplica “ativa” do ponto de vista de diferentes clientes
- Vantagem desta abordagem:
 - Load balancing dos servidores

- Cliente não pode ter acesso aos endereços das réplicas dos serviços de registro
 - Consequência: `getRegistryService` no ACS
- Connect e disconnect de receptáculos entre os serviços básicos devem ser abstraídos quanto a reconexão em caso de falha
 - Consequência: **por enquanto**, somente o SR é receptáculo do ACS
 - Como o SR pede para se conectar no ACS quando é iniciado (o que pode ser negado ou não, depende se um SR já está conectado), foi preciso colocar na idl o método *connectRegistryService*
 - Porém o ideal seria estar no connect do componente (Hugo)

- Para cada réplica
 - 1) Levantar os serviços básicos individualmente
 - Usam portas padrão
 - `run_access_control_server.sh`
 - `run_registry_server.sh`
 - Pode-se definir a porta
 - `run_access_control_server.sh --port=2090`
 - `run_registry_server.sh --port=2090`
 - 2) Levantar monitores dos serviços básicos no **mesmo host**
 - `run_ft_access_control_server_monitor.sh 2089`
 - `run_ft_registry_server_monitor.sh 2019`
 - OBS: para padronizar, ainda vou adaptar os servidores dos monitores para ficar:
 - `run_ft_access_control_server_monitor.sh --port=2089`
 - `run_ft_registry_server_monitor.sh --port=2019`

- Réplica = Componente
- Smart proxy
 - Implementação stub provida pela aplicação cliente que de forma transparente sobrescrevem os stubs padrão.
 - São meta-objetos customizáveis que podem mediar o acesso ao objeto alvo de forma flexível
- Smart component
 - Existe na literatura ??
 - Abstração criada para usarmos os serviços básicos como smart proxies em Lua



```
return {  
  hosts = {  
    ACS = { "corbaloc::localhost:2089/ACS",  
            "corbaloc::localhost:2090/ACS",  
          },  
    ACSIC = { "corbaloc::localhost:2089/IC",  
              "corbaloc::localhost:2090/IC",  
            },  
    LP = { "corbaloc::localhost:2089/LP",  
           "corbaloc::localhost:2090/LP",  
         },  
    FTACS = { "corbaloc::localhost:2089/FTACS",  
              "corbaloc::localhost:2090/FTACS",  
            },  
  },  
}
```

Lua

```
acsHostAdd-1=localhost:2019  
acsHostAdd-2=localhost:2020  
acsHostAdd-3=localhost:2021  
acsHostAdd-4=localhost:2022
```

Java


```
---
-- Habilita o mecanismo de tolerancia a falhas
--
function Openbus:enableFaultTolerance()
    log:faulthood("Mecanismo de tolerancia a falhas sendo habilitado...")
    if not self.orb then
        log:error("OpenBus: O orb precisa ser inicializado.")
        return false
    end

    if not self.isFaultToleranceEnable then
        local DATA_DIR = os.getenv("OPENBUS_DATADIR")
        local ftconfig = assert(loadfile(DATA_DIR .. "/conf/ACSFaultToleranceConfiguration.lua"))()
        local keys = {}
        keys[Utils.ACCESS_CONTROL_SERVICE_KEY] = { interface = Utils.ACCESS_CONTROL_SERVICE_INTERFACE,
                                                    hosts = ftconfig.hosts.ACS, }
        keys[Utils.LEASE_PROVIDER_KEY] = { interface = Utils.LEASE_PROVIDER_INTERFACE,
                                           hosts = ftconfig.hosts.LP, }
        keys[Utils.ICOMPONENT_KEY] = { interface = Utils.COMPONENT_INTERFACE,
                                       hosts = ftconfig.hosts.ACSIC, }
        keys[Utils.FAULT_TOLERANT_ACS_KEY] = { interface = Utils.FAULT_TOLERANT_SERVICE_INTERFACE,
                                              hosts = ftconfig.hosts.FTACS, }

        self.smartACS = SmartComponent:__init(self.orb, "ACS", keys)

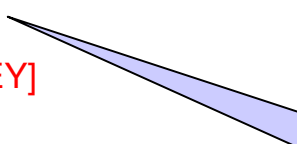
    end

    self.isFaultToleranceEnable = true
    return true
end
```

```
function __init(self, orb, compName, keys)
    smartpatch.setorb(orb)
    for key, values in pairs(keys) do
        smartpatch.setreplicas(key, values.hosts)
    end

    log:faulthood("smartpatch configurado.")
    return oop.rawnew(self, { _orb = orb,
                              _keys = keys,
                              _compName = compName, })
end
```

```
function Openbus:_fetchACS()
    local status, acs, lp, ft, ic
    if self.isFaultToleranceEnable then
        status, services = self.smartACS:_fetchSmartComponent()
    else
        status, acs, lp, ic, ft = oil.pcall(Utils.fetchAccessControlService, self.orb, self.host, self.port)
    end
    if not status then
        log:error("Erro ao obter as facetas do Serviço de Controle de Acesso." ..
            "Erro: " .. acs)
        return false
    end
    if (self.isFaultToleranceEnable and not services) or
        (not self.isFaultToleranceEnable and not acs) then
        -- o erro já foi pego e logado
        return false
    end
    if self.isFaultToleranceEnable then
        acs = services[Utils.ACCESS_CONTROL_SERVICE_KEY]
        lp = services[Utils.LEASE_PROVIDER_KEY]
        ic = services[Utils.ICOMPONENT_KEY]
        ft = services[Utils.FAULT_TOLERANT_ACS_KEY]
    end
    self.acs, self.lp, self.ic, self.ft = acs, lp, ic, ft
    local status, err = oil.pcall(self._setInterceptors, self)
    if not status then
        log:error("Erro ao cadastrar interceptadores no ORB. Erro: " .. err)
        return false
    end
    return true
end
```



Guarda referências dos
smarts proxies das
facetas

```
function _fetchSmartComponent(self)
    local status = true
    local services = {}
    local timeToTry = 0
    local stop = false
    local stops = {}
    local maxTimeToTry = 100
    local ref
    repeat
        for key,values in pairs(self._keys) do
            smartpatch.updateHostInUse(key)
        end
        for key,values in pairs(self._keys) do
            ref = smartpatch.getCurrRef(key)
            ret, stop, service = oil.pcall(Utils.fetchService, self._orb, ref, values.interface)
            if not stop then
                services = {}
                break
            else
                services[key] = service
            end
        end
        timeToTry = timeToTry + 1
        --TODO: colocar o timeToTry de acordo com o tempo do monitor da réplica
    until stop or timeToTry == maxTimeToTry
    if services == {} or not stop then
        Log:faulttolerance("[_fetchSmartComponent] Componente tolerante a falhas nao encontrado.")
        return false, nil
    end
    for key,values in pairs(self._keys) do
        smartpatch.addSmart(self._compName, services[key])
    end
    log:faulttolerance("Componente adaptado para ser um smart proxy.")
    return true, services
end
```

```
function FTACSMonitorFacet:monitor()
  Log:faulthtolerance("[Monitor SCA] Inicio")
  --variavel que conta ha quanto tempo o monitor esta monitorando
  local t = 5
  while true do
    local reinit = false
    --VERIFICA SE NAO ESTA EM ESTADO DE FALHA
    if reinit then
      local timeToTry = 0
      repeat
        if self.recConnId ~= nil then
          --DESCONECTA RECEPTACULO
        end
        Log:faulthtolerance("[Monitor SCA] Levantando Servico de Controle de Acesso...")
        --Criando novo processo assincrono
        if self.isUnix() then
          os.execute(BIN_DIR.."/run_access_control_server.sh --port=".. self.config.hostPort)
        else
          os.execute("start "..BIN_DIR.."/run_access_control_server.sh --port=".. self.config.hostPort)
        end
        -- Espera 5 segundos para que dê tempo do SCA ter sido levantado
        os.execute("sleep 5")
        local ftacsService = orb:newproxy("corbaloc: "..self.config.hostName..": "..self.config.hostPort.."/FTACS",
          "IDL:openbusidl/ft/IFaultTolerantService:1.0")

        self.recConnId = nil
        if OilUtilities:existent(ftacsService) then
          --CONECTA RECEPTACULO
        end
        timeToTry = timeToTry + 1
        --TODO: colocar o timeToTry de acordo com o tempo do monitor da réplica?
        until self.recConnId ~= nil or timeToTry == 1000
        if self.recConnId == nil then
          log:faulthtolerance("[Monitor SCA] Servico de controle de acesso nao pode ser levantado.")
          return nil
        end
        Log:faulthtolerance("[Monitor SCA] Servico de Controle de Acesso criado.")
      end
    end
    Log:faulthtolerance("[Monitor SCA] Dormindo:"..t)
    -- Dorme por 5 segundos
    oil.sleep(5)
    t = t + 5
    Log:faulthtolerance("[Monitor SCA] Acordou")
  end
end
```

Monitores

```
function ACSFacet:getRegistryService()
local acsIRecep = self:getACSReceptacleFacet()
local status, conns = oil.pcall(acsIRecep.getConnections, acsIRecep, "RegistryServiceReceptacle")
if not status then
    Log:error("[ACSFacet] Não foi possível obter o Serviço de Registro. Erro: " .. conns)
    return nil
end
if conns[1] ~= nil then
    local rgs = Openbus:getORB():narrow(conns[1].objref, "IDL:openbusidl/rs/IRegistryService:1.0")
    if Openbus.isFaultToleranceEnable then
        if not OilUtilities:existent(rgs) then
            status, services = self:getSmartRSInstance():_fetchSmartComponent()
            if not status then
                log:error("Erro ao obter as facetas do Serviço de Controle de Acesso. Erro: " .. acs)
                return nil
            end
        end
        if not services then
            -- o erro já foi pego e logado
            --desloga e desconecta o SR que está em estado de falha
            self:logout(self.registryCredential)
            return nil
        end
        rgs = services[Utils.REGISTRY_SERVICE_KEY]
    end
    return rgs
end
else
    Log:error("[ACSFacet] Não foi possível obter o Serviço de Registro. Erro: " .. conns)
    return nil
end
end
```

```
function ACSFacet:getSmartRSInstance()
if self.smartRS == nil then
    local DATA_DIR = os.getenv("OPENBUS_DATADIR")
    local ftconfig = assert(loadfile(DATA_DIR .. "/conf/RSFaultToleranceConfiguration.lua"))()
    local keys = {}
    keys[Uutils.REGISTRY_SERVICE_KEY] = { interface = Uutils.REGISTRY_SERVICE_INTERFACE,
                                           hosts = ftconfig.hosts.RS, }

    keys[Uutils.FAULT_TOLERANT_RS_KEY] = { interface =
                                           Uutils.FAULT_TOLERANT_SERVICE_INTERFACE,
                                           hosts = ftconfig.hosts.FTRS, }

    self.smartRS = SmartComponent:__init(Openbus:getORB(), "RS", keys)
end
return self.smartRS
end
```

```

function ACSFacet:connectRegistryService(registryService)
    local credential = Openbus.serverInterceptor:getCredential()
    if credential.owner == "RegistryService" then
        local acsIRecep = self:getACSReceptacleFacet()
        local status, conns = oil.pcall(acsIRecep.getConnections, acsIRecep, "RegistryServiceReceptacle")
        if not status then
            Log:error("[faulItolerance] Erro ao pegar conexoes " .. conns)
            return false
        else
            if conns[1] ~= nil then
                local rgs = Openbus:getORB():narrow(conns[1].objref, "IDL:openbusidl/rs/IRegistryService:1.0")
                if rgs:_non_existent() then
                    -- se SR nao existe, mas esta conectado, desloga (e desconecta) para poder efetuar conexao
                    self:logout(self.registryCredential)
                else
                    if credential.identifier ~= self.registryCredential.identifier then
                        --se o SR existe, esta conectado porem com outra credencial,
                        -- significa que outra replica esta tentando conectar, mas nao pode
                        --self.registryCredential = credential
                        return false
                    else
                        --se o SR existe, esta conectado e com a mesma credencial
                        -- nao faz nada
                        return true
                    end
                end
            end
        end
        end
        --OK, pode conectar
        --conectar RS no ACS: [ACS]--( 0--[RS]
        local success, conId = oil.pcall(acsIRecep.connect, acsIRecep, "RegistryServiceReceptacle",
                                         registryService )
        if not success then
            Log:error("[faulItolerance] Erro durante conexao do servico RS ao ACS.")
            Log:error(conId)
            error{"IDL:SCS/ConnectFailed:1.0"}
        end
        self.registryCredential = credential
        local entry = self.entries[credential.identifier]
        entry.component = registryServiceComponent
        local suc, err = self.credentialDB:update(entry)
        if not suc then
            Log:error("Erro persistindo referencia registry service: "..err)
        end
        return true
    end end end

```

- O estado das réplicas são atualizados em dois momentos
 - A cada requisição
 - ServerInterceptor.lua dispara uma thread para atualizar
 - Se uma entidade (credencial, oferta, etc) não for encontrada, o serviço atualiza o seu estado para garantir que realmente não existe
- Serviço de Controle de Acesso
 - Estado das credenciais
- Serviço de Registro
 - Estado das ofertas
- Gerenciamento de Serviços
 - Estados das interfaces com autorização

- ACS.lua e RS.lua

```
-----  
-- Faceta IFaultTolerantService  
-----
```

```
FaultToleranceFacet = FaultTolerantService.FaultToleranceFacet
```

```
function FaultToleranceFacet:updateStatus(self)
```

```
--Atualiza estado
```

```
...
```

```
end
```

```
public void initWithFaultTolerance(String[] args, Properties props,
    String host, int port) throws UserException {
    ....
    this.isFaultToleranceEnable = true;

    String clientInitializerClassName = FTClientInitializer.class.getName();
    props.put(ORB_INITIALIZER_PROPERTY_NAME_PREFIX + clientInitializerClassName,
        clientInitializerClassName);
    ...
}

public class FTClientInitializer extends LocalObject implements ORBInitializer {
    public void post_init(ORBInitInfo info) {
        try {
            info.add_client_request_interceptor(new FTClientInterceptor(CodecFactory.createCodec(info)));
            Log.INTERCEPTORS.info("REGISTREI INTERCEPTADOR CLIENTE TOLERANTE A FALHAS!");    }
        catch (UserException e) {
            Log.INTERCEPTORS.severe("ERRO NO REGISTRO DO INTERCEPTADOR CLIENTE TOLERANTE A
                FALHAS!", e);    }
    }
}
```

```
class FTClientInterceptor extends ClientInterceptor {
/**
 *Gerencia a lista de replicas.
 */
private FaultToleranceManager ftManager;
...
FTClientInterceptor(Codec codec) {
    super(codec);
    this.ftManager = FaultToleranceManager.getInstance();
}

public void receive_exception(ClientRequestInfo ri) throws ForwardRequest {
    ...
}
}
```

```
public void receive_exception(ClientRequestInfo ri) throws ForwardRequest {
    Log.INTERCEPTORS.info("[receive_exception] TRATANDO EXCECAO ENVIADA DO SERVIDOR!");
    String msg = "";
    boolean fetch = false;
    if (ri.received_exception_id().equals("IDL:omg.org/CORBA/TRANSIENT:1.0")) {
        fetch = true;
    } else if (ri.received_exception_id().equals("IDL:omg.org/CORBA/OBJECT_NOT_EXIST:1.0")) {
        fetch = true;
    } else if (ri.received_exception_id().equals("IDL:omg.org/CORBA/COMM_FAILURE:1.0")) {
        fetch = true;
    } else if (ri.received_exception_id().equals("IDL:omg.org/CORBA/TRANSIENT:1.0")) {
        fetch = true;
    }
    Openbus bus = Openbus.getInstance();
    ORB orb = bus.getORB();
    ParsedIOR pior = new ParsedIOR( (org.jacorb.ORB) orb, orb.object_to_string( ri.target() ));
    String key = CorbaLoc.parseKey( pior.get_object_key());
    if (key.equals(Utils.ACCESS_CONTROL_SERVICE_KEY) ||
        key.equals(Utils.LEASE_PROVIDER_KEY) ||
        key.equals(Utils.ICOMPONENT_KEY) ||
        key.equals(Utils.FAULT_TOLERANT_KEY + "ACS")){
        while (fetch){
            --BUSCA POR OUTRA REPLICA, SE ENCONTRAR LANÇA FOWARDREQUEST
        }
        System.out.println("[ACSUnavailableException] " + msg);
    } else if (key.equals(Utils.REGISTRY_SERVICE_KEY) ) {
        IRegistryService rs = bus.getAccessControlService().getRegistryService();
        throw new ForwardRequest( rs ); } }
```

```
public void receive_exception(ClientRequestInfo ri) throws ForwardRequest {
```

```
    while (fetch){
```

```
        if (ftManager.updateACSHostInUse()){
```

```
            bus.setHost(ftManager.getACSHostInUse().getHostName());
```

```
            bus.setPort(ftManager.getACSHostInUse().getHostPort());
```

```
            try {
```

```
                bus.fetchACS();
```

```
                throw new ForwardRequest( bus.getAccessControlService() );
```

```
            } catch (ACSUnavailableException e) {
```

```
                fetch = true;
```

```
                msg = e.getMessage();
```

```
            } catch (CORBAException e) {
```

```
                fetch = true;
```

```
                msg = e.getMessage();
```

```
            } catch (ServiceUnavailableException e) {
```

```
                fetch = true;
```

```
                msg = e.getMessage();
```

```
            }
```

```
        }else{
```

```
            fetch = false;
```

```
        }
```

```
    }
```

FIM