UNIFACS UNIVERSIDADE SALVADOR
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO
MESTRADO ACADÊMICO EM SISTEMAS E COMPUTAÇÃO

IGOR LEONARDO ELOY MACEDO

UMA SOLUÇÃO DE INTEGRAÇÃO DA MONITORAÇÃO EM REDES PARA EXPERIMENTAÇÃO BASEADA NA OML E PERFSONAR

Salvador
2014

IGOR LEONARDO ELOY MACEDO

UMA SOLUÇÃO DE INTEGRAÇÃO DA MONITORAÇÃO EM REDES PARA EXPERIMENTAÇÃO BASEADA NA OML E PERFSONAR

Dissertação apresentada ao Curso de Mestrado Acadêmico em Sistemas e Computação, UNIFACS Universidade Salvador, Universidade Salvador – Laureate International Universities como requisito parcial para obtenção do grau de Mestre.

Orientador: Prof. Dr. Joberto S. B. Martins.
Prof Dr. José Augusto Suruagy Monteiro.

Salvador
2014

IGOR LEONARDO ELOY MACEDO

UMA SOLUÇÃO DE INTEGRAÇÃO DA MONITORAÇÃO EM REDES PARA EXPERIMENTAÇÃO BASEADA NA OML E PERFSONAR

Dissertação aprovada como requisito parcial para obtenção do grau de Mestre em Sistemas e Computação, UNIFACS Universidade Salvador, Laureate International Universities, pela seguinte banca examinadora:

Joberto Sérgio Barbosa Martins _____
Doutor em Ciência da Computação, Université Paris VI
Universidade Salvador - UNIFACS

José Augusto Suruagy Monteiro _____
Doutor em Computer Science pela University Of California Los Angeles
Universidade Federal de Pernambuco - UFPE

Jorge Alberto Prado de Campos _____
Doutor em Engenharia e Ciência de Informação Espacial pelo National Center for Geographic Information and Analysis-NCGIA
Universidade Salvador - UNIFACS

Leobino Nascimento Sampaio _____
Doutor em Ciência da computação pela Universidade Federal de Pernambuco
Universidade Federal de Pernambuco - UFPE

Salvador, 30 de Setembro de 2014.

FICHA CATALOGRÁFICA

Elaborada pelo Sistema de Bibliotecas da UNIFACS Universidade Salvador, Laureate Internacional Universities

Macedo, Igor Leonardo Eloy

Uma solução de integração da monitoração em redes para experimentação baseada na OML e perfSONAR. / Igor Leonardo Eloy Macedo. Salvador, 2014.

97 f. : il.

Dissertação apresentada ao Curso de Mestrado em Sistemas e Computação, UNIFACS Universidade Salvador, Laureate International Universities, como requisito parcial para obtenção do grau de Mestre.

Orientador: Prof. Dr. Joberto S. B. Martins.
Co-orientador: Prof. Dr. José Augusto Suruagy Monteiro.

1. Redes para experimentação. *2. Frameworks.* 3. OML. 4. Monitoração. I. Martins, Joberto S. B., orient. II. José Augusto Suruagy, co-orient. III. Universidade Salvador – UNIFACS. IV. Título

CDD: 004.6

Dedico este trabalho a três pessoas que me acompanham com muito amor, carinho e paciência: meu pai Adelmo, minha mãe Ilma, além da minha namorada e futura esposa Elda M G Menezes.

AGRADECIMENTOS

Agradeço ao bom Deus, por me guiar e me dar forças para batalhar pelos meus sonhos.

Ao grande amigo Matheus Pacheco, que me abraçou nesta terra como um irmão, me ajudando na fase de adaptação às mudanças e contribuindo enormemente para meu crescimento pessoal e profissional.

Ao meu chefe e amigo, Luiz Carvalho, por me apoiar neste projeto, me dando condições de conciliar meu trabalho com os estudos.

Ao meu orientador, professor Dr. Joberto Martins, pela enorme paciência, interminável incentivo, e principalmente por acreditar em mim.

Ao professor Dr. Augusto Suruagy, que juntamente com o meu orientador, professor Dr. Joberto Martins, conduziram tão bem o desenvolvimento inicial das pesquisas, permitindo que alcançássemos nosso objetivo.

Aos meus amigos de mestrado, especialmente Adriano Spínola (Mano), Igor Luiz (Prof. Girafales), Marcelo Pinheiro (IéIé), Raphael Augusto (Ligeirinho) e Thiago Hohlenweger (Kareta), por todo o incentivo e apoio, pelos bons momentos de confraternização juntos e por toda a contribuição que me deram para a construção deste trabalho.

Aos meus familiares e amigos, que mesmo distantes, sempre me apoiaram e estimularam a correr atrás dos sonhos e objetivos.

"O dia em que pararmos de pensar em consciência Negra, Amarela ou Branca e passarmos a pensar na Consciência HUMANA, o racismo desaparecerá!"

Morgan Freeman.

RESUMO

A internet como é utilizada hoje, já apresenta muitas deficiências e problemas, fazendo-se necessária a elaboração e implementação de novos protocolos e serviços que possam suprir tais necessidades, cada vez maiores. Por isso, a quantidade de grupos focados em pesquisas de novas tecnologias para comunicação mundial de dados, ou seja, pesquisa em internet do futuro, do inglês *Future Internet* (FI), tem aumentado. Em paralelo, objetivando prover infraestrutura para a realização de testes das novas tecnologias de comunicação, estão as redes para experimentação, baseadas em *softwares*, providas por *testbeds* como o GENI, NitLab e FIBRE, além de outros, entregando uma infraestrutura robusta e flexível para seus usuários. Do ponto de vista arquitetural, o testbed FIBRE se apresenta como solução completa baseada na integração e/ou federação de CMFs (CMF – *Control and Management Frameworks*). Mantido por instituições espalhadas pelo mundo (Brasil, Europa, Austrália), incorpora diversas tecnologias de controle e gerência de rede, tais como OMF, ProtoGENI e Ofelia. A diversidade no uso desses gerenciadores de redes para experimentação em um mesmo *testbed*, demandou a integração da orquestração e monitoração dos dados dos experimentos. A fim de integrar os dados de monitoração, os MDIPs (*Measurement Data Integration Point*), soluções específicas para cada um dos CMFs, possuem o objetivo comum de convergir todos os dados de monitoração em um formato comum, fazendo uso dos padrões e protocolos do perfSONAR. Partindo do mesmo conceito, o OML MDIP, objeto do desenvolvimento desta dissertação, possui a capacidade de convergir os dados capturados pela OML (ferramenta de monitoração do OMF) em um formato comum, garantindo facilidade no que se refere ao acesso e à portabilidade dos dados para os usuários do testbed FIBRE. Como desenvolvido, o OML MDIP possui a capacidade de exportar os dados de um servidor OML, em um formato padronizado e compatível com o perfSONAR, visando a integração dos dados de monitoração, além de possibilitar o acesso aos dados de monitoração, de forma distribuída, por meio de sua integração com as ferramentas do iRODS.

Palavras-chave: Redes para experimentação. *Control and Monitoring Frameworks* (CMFs). OML. perfSONAR. MDIP. Monitoração. Integração. *Testbed* FIBRE.

ABSTRACT

The internet as it is used today, already has many deficiencies and problems, leading to the elaboration and implementation of new protocols and services in order to achieve a better performance. Because of this, the amount of groups focused in research on new technologies for global data communication, ie, research on Future Internet (FI), has increased. In parallel, aiming to provide infrastructure for testing of new communication technologies, there are experimental networks. They are based on software and provided by testbeds like GENI, NitLab and FIBRE, among others, delivering a robust and flexible infrastructure for their users. From the architectural point of view, the testbed FIBRE intends to be a complete solution based on the integration and/or federation of CMFs. Maintained by institutions around the world (Brazil, Europe, Australia), incorporates various control and management technologies (CMF - Control and Management Frameworks), such as OMF, ProtoGENI and Ofelia. The diversity in the use of these experimental network managers in the same testbed, demanded the integration of the orchestration and monitoring of data's experiments. In order to integrate the monitoring data, the MDIPs (Measurement Data Integration Point), a specific solutions for each CMF, have the common goal of converging all monitoring data in a common format, making use of perfSONAR standards and protocols. Based on the same concept, the OML MDIP, object of this dissertation work, has the ability to converge the data captured by OML (OMF´s monitoring tool) in a common format, ensuring ease access and data portability for FIBRE users. The OML MDIP, as developed, can export data from a OML Server, as a standardized format, compatible with perfSONAR, aiming at integration of monitoring data, in addition to enableaccess to monitoring data, in a distributed way, through its integration with iRODS tools.

Keywords: Experimental Network. Control and Monitoring Frameworks (CMFs). OML. PerfSONAR. MDIP. Integration.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIAÇÕES

AM	Aggregate Manager
App	Aplicação
AS	Authentication Service
CMF	Control and Monitoring Framewor
EC	Experiment Controller
ED	Experiment Description
FI	Future Internet
I&M	Instrumentation and Measurement
LS	Lookup Service
MA	Measurement Archive
MDIP	Measurement Data Integration Point
ML	Measurement Library
MP	Measurement Point
MS	Measurement Streams
NMWG	Network Measurements Working Group
OCF	Ofelia Control Framework
OEDL	OMF Experiment Description Language
OMF	cOntrol and Management framework
OML	Orbit Measurement Library
RC	Resource Controller
RRD	Round Robin Database
SDM	Software Defined Measurement
SO	Sistema Operacional
SOAP	Simple Object Access Protocol
SQLMA	SQL Measurement Archive
TopS	Topology Service
ToR	Top of Rack

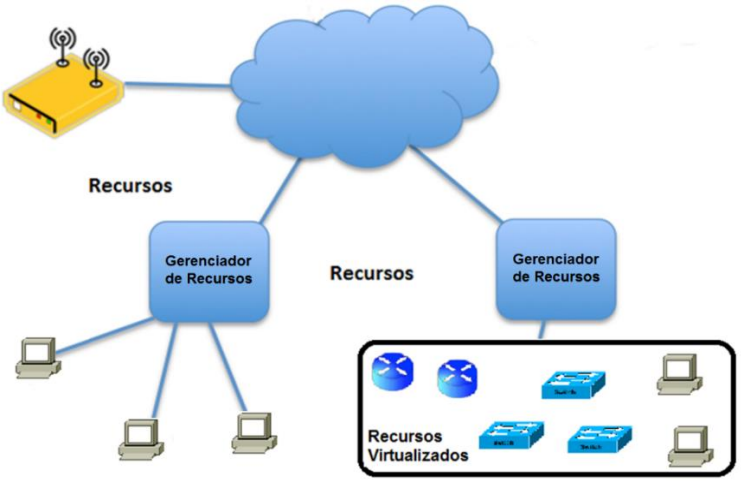
SUMÁRIO

1 INTRODUÇÃO

Com a internet se tornando cada vez mais dominante em nosso dia a dia, a exigência por algo mais seguro e eficiente também cresce de forma proporcional, dada a arquitetura atual defasada, baseada em protocolos criados há mais de 30 anos, que vêm sendo adaptados para tentar atender às demandas atuais, impulsionando a indústria e academia a desenvolverem novas tecnologias. No entanto, para que as pesquisas se transformem numa implementação confiável, faz-se necessário o uso de soluções que sejam capazes de validar de forma eficiente essas tecnologias. Dentre elas, as redes para experimentação (*testbeds*) vem se destacando, pois sua arquitetura permite simualar um comportamento complexo, equivalente à complexidade de uma grande rede, gerando resultados muito próximos aos de uma rede real.

As redes para experimentação (*testbeds*) se utilizam de recursos físicos e virtuais, sendo gerenciadas por *Control and Monitoring Frameworks (*CMFs), responsáveis pela orquestração de tais recursos de forma completa, desde alocação de processador e memória, até a quantidade de portas de rede utilizadas, além de prover ferramentas de monitoração integradas para captura e armazenamento de dados de medições, tanto da infraestrutura, como dos experimentos. A Figura 1 ilustra o que seria uma rede experimental, do ponto de vista de equipamentos disponíveis, demonstrando os recursos físicos (nó sem fio e computadores) e recursos de virtualização.

Figura 1 – Abstração de rede Experimental



Fonte: Autoria Própria (2014).

Com o objetivo de gerenciar os recursos de um *testbed*, os arcabouços de controle e gerenciamento, do inglês *Control and Management Framework* (CMF), se apresentam como

solução completa para gerência e controle dos equipamentos disponíveis, sendo, dessa forma, capazes de orquestrar um experimento, desde a sua preparação, realizando todas as configurações indicadas e sua execução, sendo inclusive, capaz de monitorar o que se deseja, durante a realização do experimento.

No contexto da monitoração dessas redes Mehani et al (2014) propôs a *Orbit Measurement Library* (OML), uma metodologia para coleta de medições, compatível com diversas tecnologias, podendo ser acoplado a qualquer CMF que se deseje. A OML é uma ferramenta fundamentada no conceito de medições definidas por software (*Software Defined Measurement (*SDM *)*), e visa a coleta, seleção e formatação de medições realizadas nessas redes, como relatado por Mehani et al (2014).

Devido às diversas características dos CMFs, muitas delas distintas (alguns focam em gerenciar experimentos em redes sem fio, outros focam na gerência de experimentos realizados em redes cabeada), atualmente existem algumas iniciativas de integração desses, objetivando o compartilhamento e otimização do uso de recursos, a exemplo do *testbed* FIBRE, mantido e provido pelo FIBRE *Project* (2014)0, constituído por instituições brasileiras e europeias. Essa integração, apesar de bastante positiva, apresenta grandes desafios, e um deles é fazer com que os CMFs conversem entre si, desde o que se refere aos recursos disponíveis, até os dados coletados de experimentos. Como cada CMF trata seus recursos de forma individual, com ferramentas exclusivas, e cada uma delas possui suas próprias metodologias, convergir essas informações num padrão comum se tornou um grande desafio.

Vislumbrando a integração das redes para experimentação, desde a configuração dos experimentos até a integração dos resultados, Pinheiro et al. (2012) propôs a FIBRE-BR I&M *Architecture* (*Instrumentation and Measurement Architecture*) para suportar a instrumentação, visualização e análise dos experimentos e resultados destes, obtidos fazendo o uso máximo das tecnologias existentes em cada CMF, sobre uma nova estrutura de rede, integrada e federada. Essa arquitetura cria um esquema de integração que transmite os dados fazendo uso de esquemas e protocolo únicos, integrando toda a comunicação.

No que se refere aos dados de medição, a FIBRE-BR I&M *Architecture* propõe o uso de uma tecnologia já bem definida para a comunicação dos dados, o perfSONAR, que de acordo com a Internet2 (2014), pode ser definido como uma infraestrutura para monitoramento de performance de redes, em ambientes federados. O perfSONAR é composto por uma série de serviços, que se comportam como uma camada intermediária entre as

ferramentas de medição e as aplicações de diagnóstico ou visualização, e podem ser adaptadas para atender as mais diversas necessidades, desde que mantenham seu comportamento padrão.

Dentro da FIBRE-BR I&M *Architecture*, foi proposto o *Measurement Data Integration Point (*MDIP), um conjunto de serviços que visa tornar cada CMF compatível com o perfSONAR, se apresentando como solução de integração dos dados de medição de cada um dos CMFs, com a arquitetura I&M do FIBRE.

Focando no *cOntrol and Management Framework (*OMF), que tem a OML como ferramenta de captura de dados de monitoração, foi proposto, desenvolvido e validado o OML MDIP, objetivo deste trabalho. Em conformidade com a arquitetura base da FIBRE-BR I&M *Architecture*, ele traz a ideia de uniformizar os dados de monitoração da OML, que a princípio podem ser definidos de forma flexível, sem padronização, garantindo um padrão para a disposição dos dados, permitindo portabilidade, além de facilitar a releitura por ferramentas que comunguem do mesmo padrão.

O capítulo dois deste trabalho se refere às redes para experimentação, com foco no *testbed* FIBRE e no OMF, justificando o motivo dessas redes e apresentando seus componentes e arquitetura de forma simplificada.

O capítulo três, fala sobre soluções de medição e monitoramento, especificamente do perfSONAR e da OML, explicando cada uma delas, demonstrando seus principais recursos e a forma como cada uma funciona.

O quarto capítulo apresenta o OML MDIP, proposta deste trabalho, destacando sua arquitetura e componentes, sua concepção e como se dá seu funcionamento.

O capítulo cinco apresenta uma prova de conceito, que objetiva comprovar o funcionamento da solução desenvolvida. Neste capítulo é apresentado todo o cenário no qual a solução foi executada, as ferramentas necessárias e os resultados alcançados.

Por fim, na conclusão do trabalho, serão relatados os principais desafios e conquistas alcançadas, além de ideias para trabalhos futuros.

2 REDES PARA EXPERIMENTAÇÃO (TESTBEDS)

A necessidade de melhorias da infraestrutura de rede da atualidade, exige que novas tecnologias sejam testadas extensivamente durante seu desenvolvimento e que os resultados se aproximem ao máximo de uma implementação confiável. As redes para experimentação (*testbeds*) vem para suprir essa demanda, visto que são plataformas de rede, com foco em experimentação, capazes de prover protótipos, emulando um ambiente muito próximo ao real, tornando possível a obtenção de resultados úteis no desenvolvimento de novos recursos, segundo o apresentado em Nicta (2014) e pelo OneLab (2014). Elas podem ser definidas como plataformas desenvolvidas para a realização de experimentos de pequena, média ou grande escala. São concebidas com fins de pesquisa, incentivando a inovação em redes, segurança, serviços e aplicações.

Atualmente existem diversas iniciativas de *testbeds*, todas visando a otimização da arquitetura atual de internet. Umas de menor escala, como por exemplo o NICTA, na Austrália, fazem uso de CMFs como o *cOntrol and Management framework* (OMF) para gerenciar pouco mais de 30 nós de rede sem fio. Já outras, de maior escala, como o *Global Enviroment for Network Innovations*, (GENI) espalhado por vários estados dos Estados Unidos, provêm uma infraestrutura virtual completa de redes e sistemas distribuídos para pesquisa e educação GENI (2014). Além desses, existem outros exemplos e um deles, especificamente o *testbed* FIBRE, será visto com maiores detalhes na seção 2.1 deste capítulo.

Os diversos CMFs que gerenciam as redes para experimentação trabalham de forma diferenciada. Alguns focam na otimização de recursos, fazendo uso de virtualização, tanto de hospedeiros (computadores), como de recursos de rede (switches, roteadores, etc), como é o caso do *Ofelia Control Framework (*OCF), baseado em um conjunto de ferramentas utilizadas para gerenciar todos os recursos disponíveis em um *testbed*, abrangendo desde a configuração dos equipamentos, até o gerenciamento do ciclo de vida do experimento, entregando partes dos recursos para cada um dos experimentadores usuários do *testbed* (OFELIA FP7 PROJECT, 2014).

Outro CMF de destaque, o ProtoGENI, funciona de forma similar ao OCF, por se basear na entrega dos recursos disponíveis, de forma parcial, para cada experimentador, permitindo um melhor uso do conjunto de recursos existente. Este CMF foi desenvolvido dentro do contexto do projeto GENI e concebido como um projeto que objetivava a

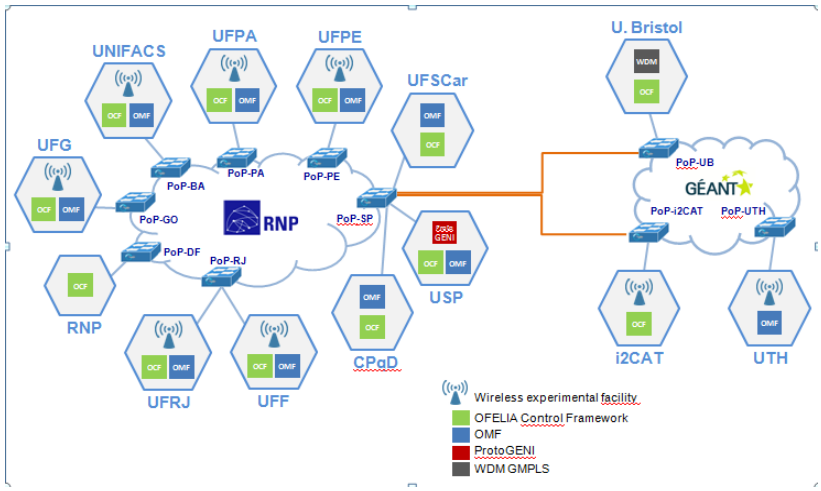
integração em larga escala de sistemas protótipos do GENI, de forma a prover uma infraestrutura que atendesse os pré-requisitos: funcionar sobre um *backbone* de alta velocidade, ter um agregado divisível em partes, ou seja, um recurso poder ser utilizado em mais de um experimento simultaneamente sem que o uso de uma parte impacte na outra, ter seus componentes programáveis, além de permitir múltiplas redes (PROTOGENI, 2014).

Há também CMFs que focam no gerenciamento de recursos de redes sem fio, que por não serem passíveis de emulação ou simulação, acabam se dedicando ao gerenciamento de equipamentos físicos, como é o caso do OMF, descrito com maiores detalhes na seção 2.2 deste capítulo, já que serviu de base para o desenvolvimento desta dissertação.

2.1 O *TESTEBED* FIBRE

O *testebed* FIBRE é o resultado de uma parceria entre o Brasil e países europeus que, de acordo com o portal do FIBRE (2014), tem como objetivo principal prover um espaço distribuído entre os países envolvidos com projetos de *Future Internet* (FI), para a realização de pesquisa experimental em uma infraestrutura de rede e aplicações distribuídas. Uma topologia genérica é apresentada na Figura 2, na qual se pode observar a presença de instituições participantes do *testbed* FIBRE.

Figura 2 – Topologia testbed FIBRE



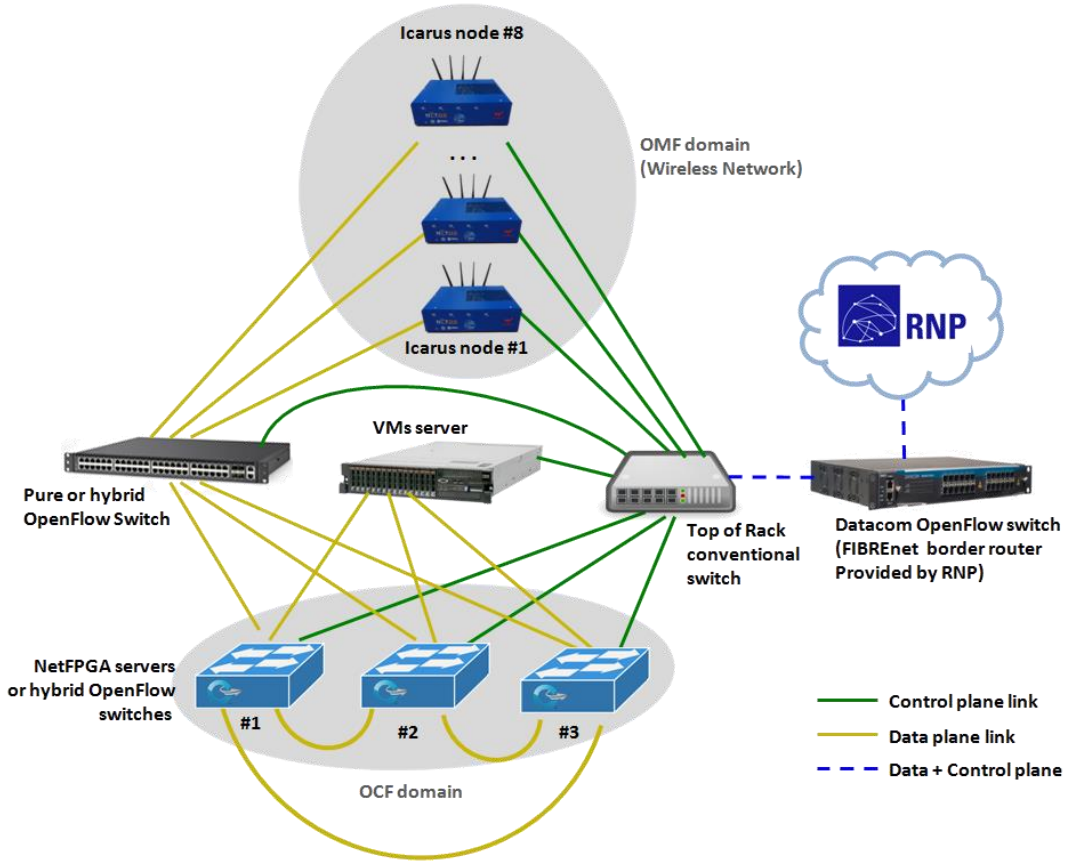
Fonte: Suruagy e Martins (2013).

Na Figura 2, cada uma das células representa uma ilha[1], sendo elas dispostas no Brasil e em países europeus. As ilhas situadas no Brasil são dotadas de diversos equipamentos, todos

[1] Instituições que provêm infraestrutura física para os recursos utilizados pelos testbeds.

disponíveis para serem utilizados nos experimentos, ilustrados na Figura 3, dentre eles: nós Ícarus (equipamento e rede sem fio, utilizado principalmente e pelo OMF), switches ethernet *Top Of Rack* (TOR), servidores netFPGA (utilizados para virtualização de redes, com *OpenFlow*[2]), switches *OpenFlow* e servidores robustos para hospedagem de hospedeiros virtualizados e outros. Tanto as ilhas brasileiras, como as europeias, podem ou não dispor de todos os equipamentos relacionados, cada uma delas apresentando suas particularidades em topologia e disponibilidade.

Figura 3 – Equipamentos testbed FIBRE



Fonte: Fibre Wiki (2014).

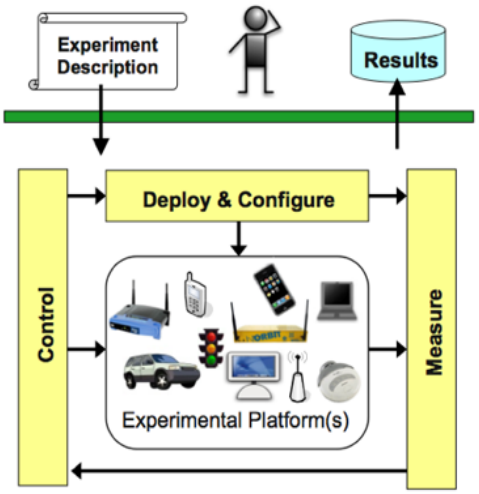
Funcionalmente, todos os recursos supracitados compõem uma infraestrutura compartilhada entre três CMFs (OFELIA, ProtoGENI, OMF), provendo uma solução completa para orquestração dos experimentos, disponibilizando recursos de rede programáveis, permitindo que experimentadores possam reazilar uma gama de experimentos em redes sem fio e cabeadas, muito grande.

[2] OpenFlow é um protocolo de comunicação de redes de computadores, que tem por base a implementação de uma camada de software para gerenciamento universal dos equipamentos de rede.

2.2 OMF – CONTROL AND MANAGEMENT FRAMEWORK

O OMF é um CMF que, por conceito, é composto por um conjunto de ferramentas capazes de instrumentar experimentos e gerenciar redes para experimentação. De acordo com o Nicta (2014), o OMF pode ser definido a partir do ponto de vista do usuário do sistema, como também analisando a visão do administrador da rede. No primeiro caso, conforme retratado pela Figura 4, o OMF provê um conjunto de ferramentas capazes de descrever, instrumentar, executar e coletar resultados dos experimentos. Já no segundo caso, o OMF é um conjunto de serviços que objetivam gerenciar e operar os recursos do sistema de forma eficiente.

Figura 4 - OMF - Visão do usuário



Fonte: Nicta (2014).

Apesar de ter sido criado inicialmente para o projeto ORBIT, o OMF foi concebido baseado na filosofia de *testbeds* genéricos, ou seja, que suportam recursos com as mais diversas tecnologias e de acordo com o Nicta (2014), ele está amadurecendo para suportar e apoiar cada vez mais novos recursos e tecnologias.

Diferentemente de outros CMFs, o OMF teve sua criação focada na mobilidade, ou seja, foi desenvolvida uma solução com a ideia de que os experimentos fossem realizados em redes sem fio ou até fora dos *testbeds*, fazendo uso de dispositivos móveis. Apesar disso, como supracitado, o OMF está amadurecendo e atualmente já suporta experimentos com redes físicas, ou seja, cabeadas, e até virtualizados, fazendo uso de switches *OpenFlow* e até máquinas virtuais.

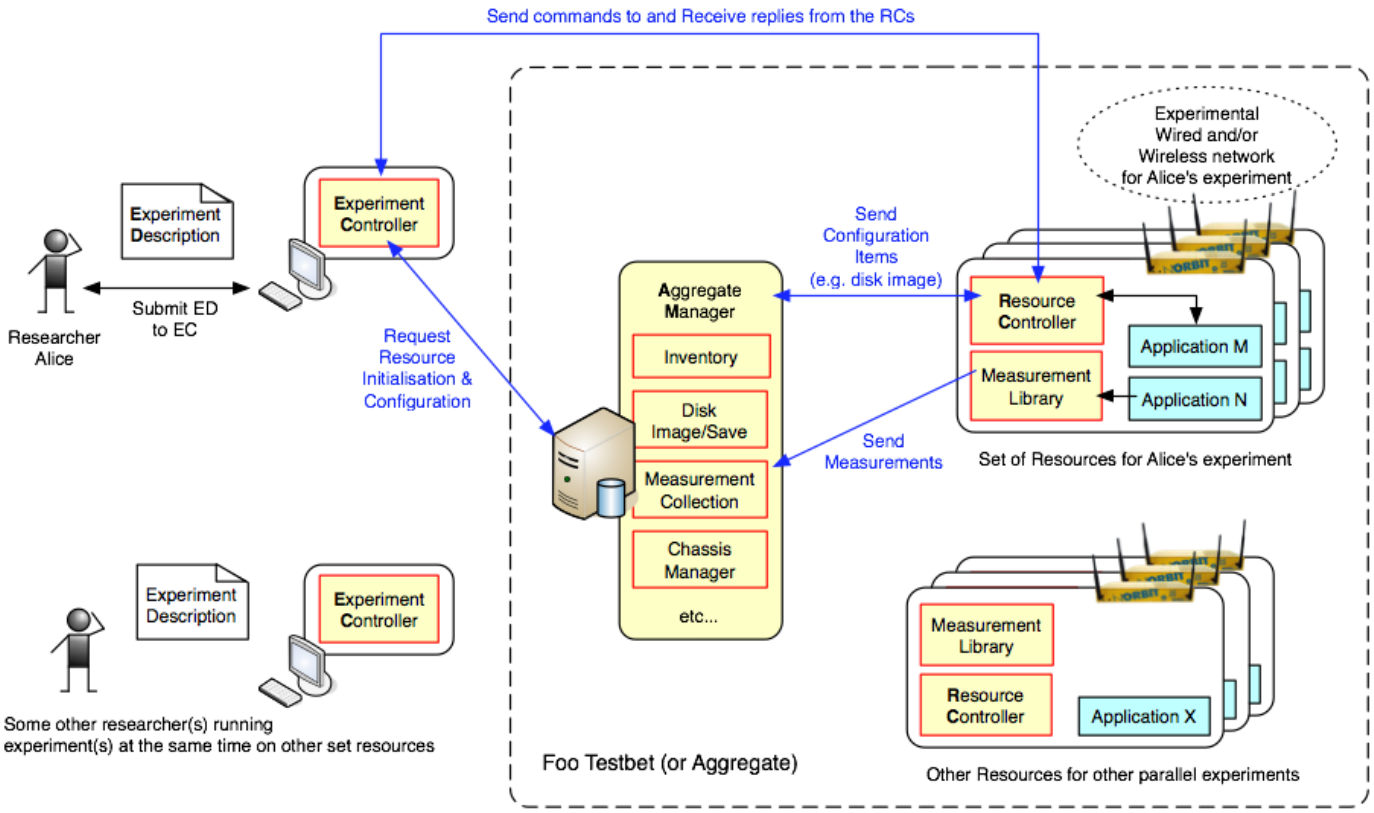
O OMF é dotado de uma arquitetura com componentes que fornecem uma série de serviços capazes de gerenciar de forma eficiente todos os recursos de um *testbed*, ou seja, ele

é capaz de inicializar/reinicializar hospedeiros, reportar os seus status, instalar novos sistemas, dentre outras funcionalidades.

2.2.1 Arquitetura do OMF

A Figura 5 ilustra a arquitetura do OMF, cujo foco é quase que exclusivo em redes sem fio. Ela ilustra os componentes chave, essenciais para o funcionamento do OMF, além de todo o fluxo de um experimento, desde sua elaboração, através da *Experiment Description*, até a comunicação do *Aggregate Manager* com os recursos.

Figura 5 - Arquitetura do OMF



Fonte: Nicta (2014).

Os componentes do OMF podem ser categorizados como ferramentas de Gerência, Controle e Medição, agrupados da seguinte forma:

- **Ferramentas de Gerência:**
  - *Aggregate Manager* (AM): sistema capaz de gerenciar todos os recursos da rede experimental, entregando os recursos ao experimentador, quando requisitado. Um conjunto de recursos é chamado de agregado.

- **Ferramentas de Controle:**
  - *Experiment Controller* (EC): agente responsável pelo interfaceamento entre o usuário (experimentador) e os componentes do OMF. Este componente permite a execução de comandos como iniciar um experimento, solicitar recursos do testbed, entre outros;
  - *Experiment Description* (ED): todos os componentes supracitados falam uma mesma "língua", a ED, ou seja, ela é um conjunto de instruções, que são utilizadas para definir as ações a serem executadas num experimento. O OMF utiliza a OEDL (OMF *Experiment Description Language*), exemplificada na Figura 6, e é por meio dela que há a indicação dos recursos a serem utilizados e suas respectivas configurações, as aplicações utilizadas, bem como aplicações e ações necessárias para o experimento.

Figura 6 - OEDL: Definição de um hospedeiro

```
defGroup('Sender', "omf.nicta.node2") do |node|
  node.addApplication("test:app:otg2") do |app|
    app.setProperty('udp:local_host', '192.168.0.2')
    app.setProperty('udp:dst_host', '192.168.0.3')
    app.setProperty('udp:dst_port', 3000)
    app.measure('udp_out', :interval => 3)
  end
  node.net.w0.mode = "adhoc"
  node.net.w0.type = 'g'
  node.net.w0.channel = "6"
  node.net.w0.essid = "helloworld"
  node.net.w0.ip = "192.168.0.2"
end
```

Fonte: Nicta (2014).

  - *Resource Controller* (RC): componente interpretador de instruções recebidas a partir do AM e/ou do EC. Fica instalado no recurso[3], sendo capaz de traduzir os comandos recebidos em ações, como por exemplo, iniciar uma aplicação, rebootar o recurso, etc;

- **Ferramentas de Medição:**
  - *Measurement Library* (ML): biblioteca de códigos para desenvolvimento, criada baseada na linguagem C, que deve ser utilizada em todos os recursos do experimento, que se deseje coletar

[3] No contexto deste trabalho, é um recurso de rede instalado num *testbed* gerenciado pelo OMF.

dados de medição. Esta biblioteca, é o núcleo da OML, que será detalhado no capítulo 3 deste trabalho.

A arquitetura apresentada na Figura 5 apresenta um modelo de gerência centralizado, pelo qual é possível compartilhar recursos da infraestrutura de forma que experimentos possam ser executados paralelamente. Entretanto, essa mesma arquitetura se mostra limitada no sentido que não há a possibilidade de compartilhamento de recursos, ou seja, uma vez que um recurso é alocado a um experimento, ele estará disponível somente àquele.

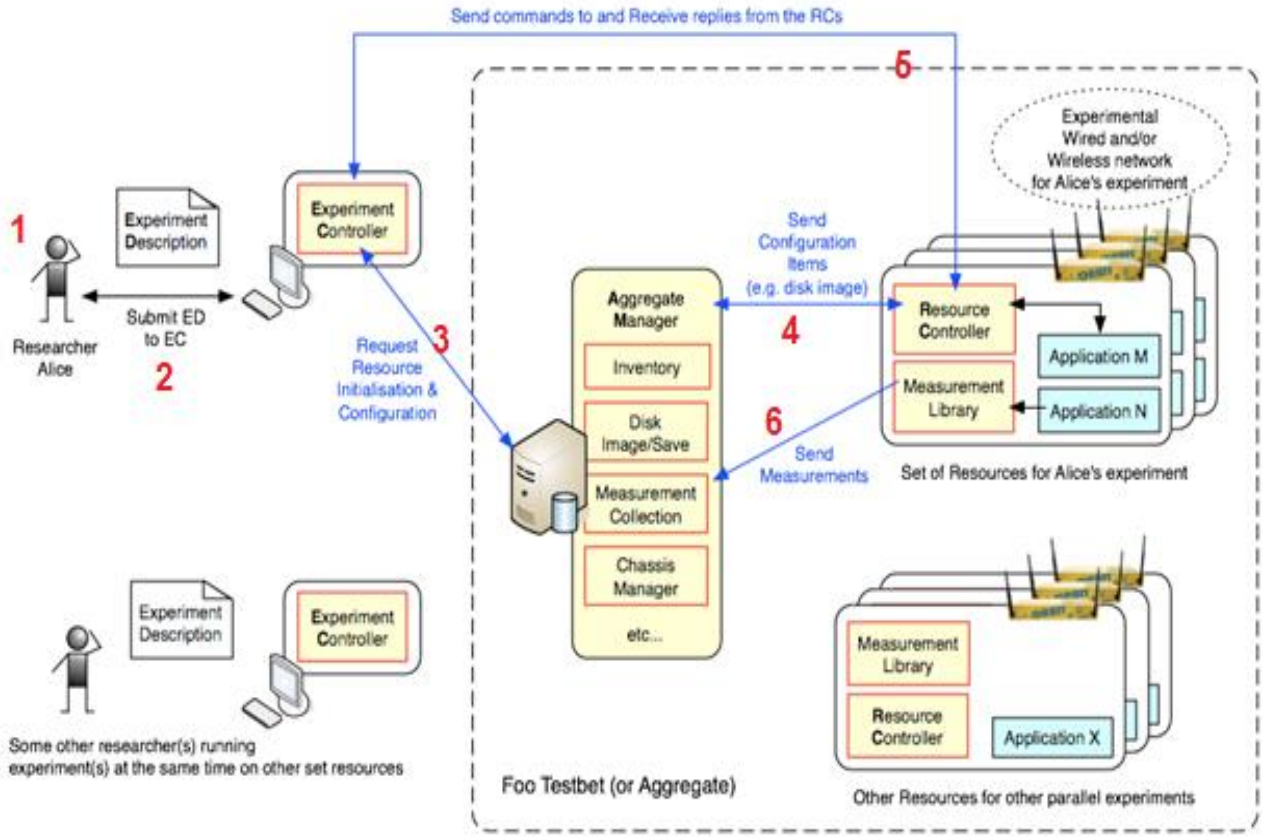
Do ponto de vista arquitetural, cada experimento segue um fluxo único e obrigatório, conforme descrito abaixo:

- O experimentador deve descrever como deseja que seu experimento seja executado, fazendo uso da ED e a submete ao EC;
- O EC interpreta o que foi descrito pela ED e faz todas as requisições necessárias para o AM, que inicializa e configura todos os recursos solicitados;
- Quando os recursos estão prontos, o EC dispara a inicialização do experimento. Os RCs de cada um dos recursos interpretam os comandos recebidos, orquestrando o experimento como foi definido na ED;
- Enquanto o experimento é executado, as aplicações enviam dados de medição para o servidor OML, que as armazena em um banco de dados – BD, para acesso posterior.

2.2.2 Ciclo de vida de um experimento no OMF

Do ponto de vista do experimentador, é muito importante entender como o experimento é criado e executado. A Figura 7 traz a arquitetura do OMF orientada ao fluxo de um experimento, no qual inicialmente **(1)** o experimentador deverá desenvolver ou modificar uma aplicação capaz de ser executada em um sistema operacional (SO), normalmente Linux, nos recursos desejados do *testbed* gerenciado pelo OMF. Havendo interesse em coleta de dados de medição, o código deverá ser modificado de modo a suportar a ML, para que os dados possam ser coletados pela OML, como será demonstrado no capítulo 3 deste trabalho.

Figura 7 – Fluxo de um experimento do OMF



Fonte: Autoria própria, adaptado de Nicta (2014).

Posteriormente, é necessário **(2)** definir como será o experimento, fazendo uso da **ED**, que descreve os recursos necessários para o experimento e suas respectivas configurações. Além disso, a ED descreverá as ações e sua ordem de execução. Como dito anteriormente a ED é descrita através da OEDL.

Definido o experimento, deve-se **(3)** executá-lo por meio do EC e para tal, os seguintes passos são necessários:

- Ter acesso a um *testbed* baseado no OMF, sendo necessário ter um *login* e uma senha;
- Reservar os recursos necessários. O esquema de reserva de recursos varia para cada testbed. Segundo Nicta (2014), o WinLab faz essas reservas por horários, objetivando o compartilhamento dos recursos entre todos os usuários;
- Se certificar que as aplicações para o experimento e a ED são acessíveis pelo testbed (o testbed só acessa esses arquivos localmente, ou seja, talvez seja necessário copiar as aplicações e arquivos de configuração, para o testbed);

- Rodar o experimento com o uso do OMF Tools, em especial o aplicativo omf (omf [subcomando argumento] [arquivo da ED]). O Quadro 1 exibe a lista de subcomandos existentes no app omf;

Quadro 1 - Lista de subcomandos do app omf

help	Exibe a ajuda
exec	Executa o experimento
load	Carrega uma imagem (disk image) para os nós
save	Salva uma imagem nos nós
tell	Liga e desliga os nós
stat	Retorna o status dos nós

Fonte: Nicta (2014).

Após a submissão da ED por meio do EC, o AM **(3)** recebe a requisição de instalações e requisições, **(4)** repassando-as a um ou mais RCs. Opcionalmente o experimentador pode interagir com os recursos **(5)** por meio do RC.

Caso o experimento faça uso da OML, após sua execução, as medições realizadas durante ele deverão ser **(6)** coletadas para análise. Durante um experimento, um OML *Client* coleta informações e as envia para a OML *Collection Server* que armazena as informações em uma base de dados com o ID igual ao do experimento, ou definida pelo experimentador. É através deste ID que a base de dados pode ser acessada para posterior consulta.

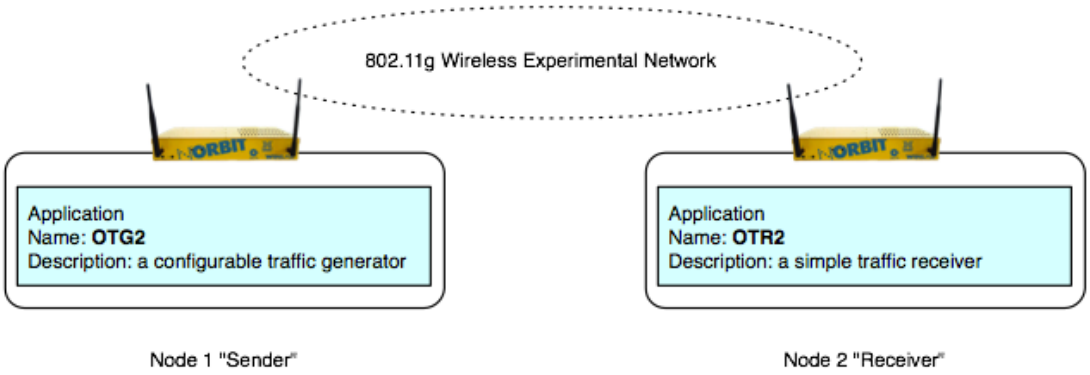
2.2.3 Exemplo de experimento OMF

A fim de obter um melhor entendimento de toda a solução OMF/OML, esta seção se propões a demonstrar como o OMF funciona. Neste exemplo, será utilizado um exemplo básico, executado no *testbed* do NICTA, que objetiva demonstrar como desenvolver, executar e obter os resultados de um experimento realizado no OMF.

2.2.3.1Cenário

A Figura 8 ilustra o cenário do experimento a ser executado. Nela, é possível verificar que existem 2 equipamentos para comunicação de redes sem fio, sendo um deles transmissor e o outro receptor. A aplicação utilizada no transmissor é o OTG2, um gerador de tráfego UDP. Já no receptor, utilizou-se o OTR2, um receptor de tráfego. A comunicação entre os dois equipamentos se dá através do ar, ou seja, fazendo uso de redes sem fio.

Figura 8 - Cenário do experimento



Fonte: Nicta (2014).

O primeiro passo para utilizar um *testbed* é obter uma conta de acesso. Com o acesso ao testbed é necessário carregar a aplicação a ser utilizada no experimento, utilizando o comando "omf load". Neste exemplo, as aplicações (OTG e OTR) já estão instaladas no sistema base do recurso, tornando o comando "omf load" desnecessário. Após a carga da aplicação, para inicializá-la é necessário submeter a ED da aplicação para o EC.

A

Figura 9, exemplifica a ED para o experimento em questão, escrita na linguagem OEDL. Nela observa-se detalhes descritivos sobre os recursos requeridos e o conjunto de ações a serem executadas para o experimento. As linhas 1 e 15 definem novos grupos de recursos, denominados de "Sender" e "Receiver" identificados pelos identificadores únicos "1,2" e "1,3" respectivamente. Nas linhas de 2 a 7 são feitas definições a respeito da aplicação são feitas. Na linha 2 uma aplicação de nome "test:app:otg2" é associada a este grupo de recursos, ou seja, quaisquer recursos que estejam vinculados a este grupo terão a aplicação especificada instalada. Na linha 3 a propriedade "local_hospedeiro" da aplicação foi definida com o número de ip 192.168.0.2 indicando o endereço do remetente. As linhas 4 e 5 indicam o endereço e porta do destinatário. A linha 6 diz que a aplicação fará medições no ponto de medição especificada como "udp_out".

Figura 9 - Definição de Experimento OMF (ED com OEDL)

```
1.  defGroup('Sender', [1,2]) {|node|
2.    node.addApplication("test:app:otg2") {|app|
3.      app.setProperty('udp:local_hospedeiro', '192.168.0.2')
4.      app.setProperty('udp:dst_hospedeiro', '192.168.0.3')
5.      app.setProperty('udp:dst_port', 3000)
6.      app.measure('udp_out', :interval => 3)
7.     }
8.     node.net.w0.mode = "adhoc"
9.     node.net.w0.type = 'g'
10.    node.net.w0.channel = "6"
11.    node.net.w0.essid = "helloworld"
12.    node.net.w0.ip = "192.168.0.2"
13.  }
14.
15.  defGroup('Receiver', [1,3]) {|node|
16.    node.addApplication("test:app:otr2") {|app|
17.      app.setProperty('udp:local_hospedeiro', '192.168.0.3')
18.      app.setProperty('udp:local_port', 3000)
19.      app.measure('udp_in', :interval => 3)
20.    }
21.    node.net.w0.mode = "adhoc"
22.    node.net.w0.type = 'g'
23.    node.net.w0.channel = "6"
24.    node.net.w0.essid = "helloworld"
25.    node.net.w0.ip = "192.168.0.3"
26.  }
27.
28.  whenAllInstalled() {|node|
29.    info "This is my first OMF experiment"
30.    wait 10
31.    allGroups.startApplications
32.    info "All my Applications are started now..."
33.    wait 30
34.    allGroups.stopApplications
35.    info "All my Applications are stopped now."
36.    Experiment.done
37.  }
```

Fonte: Nicta (2014).

As linhas de 8 a 12 indicam propriedades para o grupo “sender”. A linha 8 diz que a interface wireless “w0” dos recursos deste grupo ficarão no modo “adhoc”. As linhas 9 e 10

indicam que a mesma interface transmitirá utilizando o protocolo 802.11g através do canal 6. O SSID da rede será “helloworld” e o IP da mesma interface será 192.168.0.2.

As linhas de 15 a 26 indicam as mesmas configurações, porém para o grupo de recursos “receiver”.

As linhas de 28 a 37 descrevem as ações. O OMF utiliza o modelo de máquinas de estado para descrever as ações a serem executadas num experimento, o que quer dizer que ações estão associadas a estados, e são executadas quando os estados são alcançados. Um estado no OMF pode ser caracterizado por diferentes condições, como por exemplo, o estado de todos os recursos serem ligados, ou quando uma variável atingir um determinado valor, entre outros. No experimento em questão, há apenas uma ação, nomeada de “whenAllInstaled”, na linha 28, que basicamente verifica quando todos os nós estão ligados e todas as aplicações estão instaladas. As linhas 29, 32 e 35 lançam no console algumas mensagens indicando o status do experimento. A linha 30 manda o EC parar por 10 segundos. Vale ressaltar que esta paralisação é recomendada para garantir que todos os recursos do testbed alocados para o experimento sejam configurados. A linha 31 diz aos grupos para inicializar as aplicações associadas a eles. Na linha 33, observa-se uma nova solicitação de pausa, por 30 s, garantindo que o tráfego enviado do transmissor chegará ao receptor. A linha 34 manda que os grupos parem a execução de suas aplicações. A linha 36 encerra o experimento limpando todas as configurações e ações realizadas anteriormente.

Uma vez que o experimento foi definido, faz-se necessário certificar os seguintes itens:

- Ter os recursos reservados por um determinado intervalo de tempo, de forma a garantir que o experimento executado não sofrerá interferências externas;
- Estar com o console do *testbed* acessível;
- Ter carregado a aplicação para o *testbed*;

Após os passos supracitados, é necessário criar uma nova instância do EC, responsável pela execução do experimento. Isto é feito com uma chamada ao aplicativo “omf”, passando o subcomando “exec”. Este comando deverá receber um parâmetro como entrada, que será a ED, que estará em um arquivo texto simples com uma extensão. Seguindo os padrões do OMF, todo arquivo de ED deverá ter a extensão “.rb”. Dessa forma, a chamada ao omf ficará da seguinte maneira: “omf exec myED.rb”, sendo “myED.rb” o arquivo de descrição do experimento.

Após a conclusão da execução do experimento, graças ao uso da OML, é possível obter dados a respeito dele. Os dados coletados estão armazenados em um banco de dados

pertencente a OML *Measurement Collection Server*. Cada experimento executado cria um novo banco de dados com o mesmo nome do identificador do experimento.

Cada instalação do OMF possui um ou mais AM, que provêm uma série de recursos para gerenciamento, dentre eles um portal no qual também é possível acessar os resultados.

3 SOLUÇÕES DE MONITORAÇÃO/MEDIÇÃO

As soluções para experimentação em redes, como os *testbeds*, são muito eficazes, visto que conseguem resultados muito próximos aos de um ambiente de produção. Porém, as redes para experimentação não seriam tão eficientes, se não possuíssem ferramentas capazes de medir e armazenar os dados de um experimento e de sua infraestrutura. Para isso os *testbeds* proveem (nativamente ou não) soluções para atender essas necessidades que, como relatado pelo Pinheiro et al. (2012) é o caso da OML, Instolls[4] e Lamp[5] para medição a nível de experimentos e ZenOSS[6] e perfSONAR para medições de infraestrutura.

Cada solução de monitoramento, possui sua própria metodologia de obtenção dos dados, fazendo uso de protocolos com métricas já definidas ou sendo capaz de obter dados completamente arbitrários. Algumas fazem uso do SNMP ou se utilizam da arquitetura do perfSONAR. Já outras, como é o caso da OML, possui um esquema proprietário. Dentro deste capítulo, serão explorados o perfSONAR (na próxima seção) e a OML (seção 3.2), pois a união de ambos gerou o desenvolvimento para o OML MDIP, solução proposta nesta dissertação. Enquanto que o primeiro serviu como base para a integração dos dados, o segundo foi utilizado devido à sua extrema flexibilidade.

3.1 PERFSONAR

De acordo com a Internet2 (2014), o perfSONAR é uma solução que provê infraestrutura para monitoramento de performance de rede, servindo para facilitar a resolução de problemas ponto-a-ponto, em caminhos que cruzem diversas redes, por meio de um conjunto de serviços que entregam medições de desempenho em ambientes federados. Possui uma arquitetura baseada em serviços, que se comunicam uns com os outros utilizando protocolos bem definidos.

O perfSONAR é patrocinado por um consórcio de instituições que buscam uma solução para análises em intra e inter-redes. Com esse fim, vários pacotes de software (serviços) foram produzidos, a fim de conceber um framework de análise de performance

---

[4] Instools (*Instrumentation Tools*) é um *framework* de instrumentação e medição de redes criado com o objetivo de facilitar a publicação da infraestrutura de medição de redes num *slice*

[5] Lamp (*Leveraging and Abstracting Measurements with perfSONAR*) é um sistema de monitoramento baseado no perfSONAR

[6] Zenoss é uma solução de monitoração de ativos de rede

completo, com seus módulos desenvolvidos por cada um dos parceiros. Para tanto, um protocolo de comunicação baseado em *Simple Object Access Protocol* (SOAP), para comunicação de dados, e no *Network Measurements Working Group (*NMWG), para formatação dos dados, foi definido, permitindo que qualquer um pudesse escrever seu serviço. Ambos as tecnologias são descritas com maiores detalhes neste capítulo.

3.1.1 NMWG – Network Measurements Working Group

Analisar a performance de uma rede requer um extenso e exaustivo trabalho de execução prévia de ferramentas capazes de obter informações sobre o desempenho da rede. Essas ferramentas, apesar de funcionarem de maneira parecida, acabam entregando as informações de formas diversificadas. Devido a isso, percebeu-se a necessidade de unificação dessas informações, de forma que o resultado final fosse sempre o mesmo. Os esquemas NMWG foram implementados a fim de resolver questões como essas, sendo este o principal motivador para sua aplicação no desenvolvimento do OML MDIP.

Baseado no XML, os esquemas NMWG foram criados com a finalidade de viabilizar uma metodologia de troca de dados formatados. De acordo com Zurawski, Swany e Gunter (2006), apesar dessa abordagem não ser nova, vide o SNMP[7], a ideia desses esquemas é utilizar identificadores mais amigáveis, diferentes, por exemplo, dos OID's[8] do SNMP, fazendo uso de *WebServices* (SOAP), facilitando possíveis extensões, permitindo que novas métricas de medição possam ser adicionadas de forma mais simples.

Da forma que foi concebido, toda a definição dada pelo NMWG se equivale a um protocolo de comunicação, no qual há a formalização de todo o conteúdo da mensagem, tanto de requisição, como de resposta. Um de seus destaques, é sua capacidade de separar informações mais dinâmicas, identificadas com um elemento XML "Data" (os dados propriamente ditos), de informações que quase não sofrem alteração, identificados como "Metadata" (seria um metadado). Por exemplo, para uma solicitação de *traceroute*[9], os dados

---

[7] *Simple Network Management Protocol* – SNMP – é um protocolo de rede utilizado para a gerência de seus recursos.

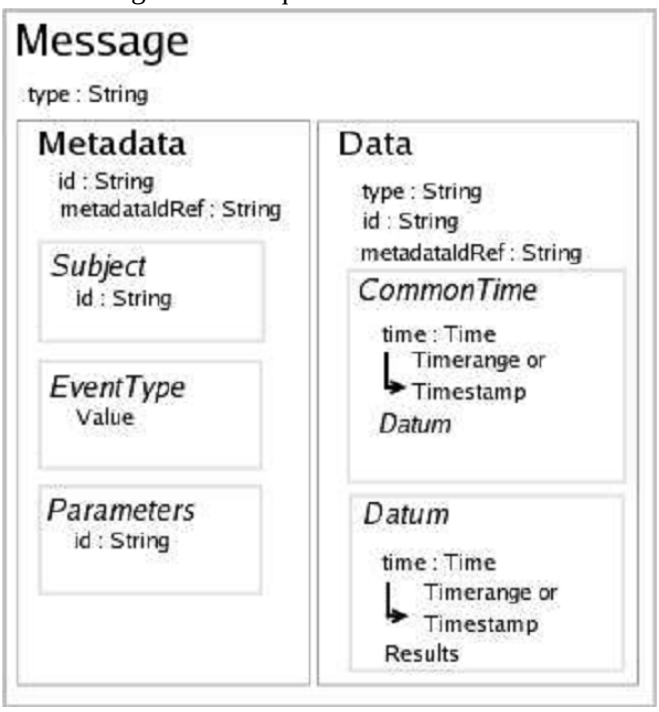
[8] *Object Identifiers* – OID – são identificadores, previamente especificados, que representarão objetos a serem monitorados via SNMP.

[9] Ferramenta de diagnóstico de rede que serve para exibir a rota (caminho) de um pacote, desde a sua origem, até o seu destino.

seriam equivalentes aos saltos efetuados pelo pacote. Já o metadado, seriam os IPs dos hospedeiros de origem e destino.

A formalização do NMWG é derivada do RELAX-NG[10], tendo sua formatação dada por esquemas NMWG, formatados com o uso do XML, definidos em arquivos ".rnc". Todos os esquemas existentes, tendem a derivar de um esquema base. O formato padrão dos esquemas NMWG está representado graficamente pela Figura 10, tido como base abstrata para todos os outros.

Figura 10 - Esquema base do NMWG



Fonte: Zurawski, Swany e Gunter (2006).

O esquema retratado na Figura 10, é traduzido para um esquema abstrato em seu formato adequado, na

Figura 11. Toda mensagem de um esquema NMWG deve possuir um elemento METADATA, que descreverá o dado de forma exclusiva, podendo vir acompanhado dos elementos *Subject, EventType e Parameters*, sendo todos apresentados ou apenas alguns deles e descritos com maiores detalhes a seguir. Todo elemento de Metadata precisa de um "id", que opcionalmente pode vir acompanhado de um metadataId (também representado como metadataIdRef), utilizado para fazer referência a outra seção de Metadata a geração de *links*, a fim de reduzir o overhead de armazenamento.

[10] RELAX-NG é uma linguagem baseada em XML, criada com a finalidade de se tornar um vocabulário XML padrão para a derivação de outras linguagens, como o NMWG, por exemplo.

Figura 11 – Definição de esquema retratado na Figura 10

```
namespace nmwg = "http://ggf.org/ns/nmwg/2.0/"
Metadata =
  element nmwg:metadata {
    attribute id { xsd:string } &
    attribute metadataIdRef { xsd:string }? &
    Subject &
    EventType? &
    Parameters?
  }
Subject =
  element nmwg:subject {
    attribute id { xsd:string }
  }
EventType =
  element nmwg:eventType {
    text?
  }
Parameters =
  element nmwg:parameters {
    attribute id { xsd:string }
  }

Data =
  element nmwg:data {
    element id { xsd:string } &
    element metadataIdRef { xsd:string } &
    (
      CommonTime? &
      Datum*
    )
  }
CommonTime =
  element nmwg:commonTime {
    Time &
    Datum*
  }
Datum =
  Time
}
```

Fonte: Zurawski, Swany e Gunter (2006).

As mensagens NMWG também possuem o elemento DATA, que deve vir acompanhado de um ID, identificador único do dado, e fazer referência a um METADATA, por meio do elemento *metadataidref*. Os valores dos dados requisitados numa mensagem, vêm sempre delimitados pelo elemento DATUM, dentro do DATA.

Analisando a

Figura 11, a seção do METADATA pode ser composta por:

- *Subject*: único dos componentes exigidos, é responsável por conter a descrição da entidade. Por exemplo, numa comunicação entre dois hospedeiros, poderá haver duas seções *subject*, uma para o hospedeiroA e outra para o hospedeiroB;
- *EventType*: descreve o tipo de medição para aquela seção. Por exemplo, num experimento que utilize o ping para testes de latência, poderá utilizar algo parecido como "latência_ICMP" para representar o tipo do evento.
- *Paramenters*: informa os parâmetros a serem utilizados na query NMWG. Por exemplo, argumentos de linha de comando, podem ser informados nesta seção.

Visando atender às mais diversas demandas, o NMWG, representado de forma simplificada Quadro 2, permite que seus esquemas sejam estendidos. Os XML *namespaces*[11] permitem que extensões derivadas de um mesmo esquema possam coexistir. Além de representar as extensões, os *namespaces* podem indicar a versão dos esquemas.

Quadro 2 - Definição simplificada de esquema com Exemplo do esquema definido

Esquema NMWG – *Utilization* (simplificado)	**Instancia XML baseada no esquema *utilization***

namespace utilization =
"http://ggf.org/ns/nmwg/characteristic/util/1.0/"
namespace nmwgt =
"http://ggf.org/ns/nmwg/topology/2.0/"
include "nmbase.rnc" {
Subject = UtilizationSubject
}
UtilizationSubject =
element utilization:subject {
attribute id { xsd:string } &
Interface?
}
Interface =
element nmwgt:interface {
element nmwgt:ipAddress { {
xsd:string &
attribute type { xsd:string }
}? &
element nmwgt:hospedeiroName { xsd:string }? &
element nmwgt:ifIndex { xsd:string }? &
element nmwgt:type { xsd:string }? &
element nmwgt:direction { xsd:string }?
	<nmwg:metadata id="1">
	<utilization:subject id="2">
	<nmwgt:interface>
	<nmwgt:ipAddress type="v4">
	128.4.133.163
	</nmwgt:ipAddress>
	<nmwgt:hospedeiroName>
	moonshine.pc.cis.udel.edu
	</nmwgt:hospedeiroName>
	<nmwgt:ifName>eth0</nmwgt:ifName>
	<nmwgt:ifIndex>2</nmwgt:ifIndex>
	</nmwgt:interface>
	</utilization:subject>
	<nmwg:eventType>ifInOctets</nmwg:eventType>
	</nmwg:metadata>

[11] *NameSpaces* são URIs (*Uniform Resource Identifier*) específicas, similares às URLs (*uniform Resource Location*), pois também fazem uso da string de endereçamento padrão da internet, por exemplo, http://www.google.com

}	

Fonte: Zurawski, Swany e Gunter (2006).

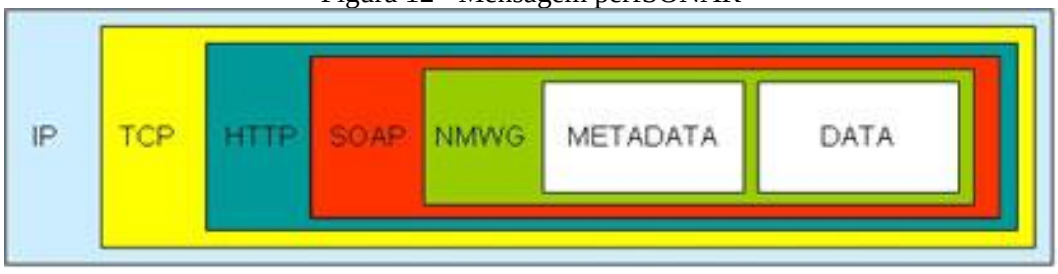
O Quadro 2, traz um exemplo simplificado da definição de um esquema real., que padroniza como informações de interface de rede serão formatadas para serem enviadas nas mensagens. A segunda coluna desse quadro exemplifica como seria uma mensagem XML instanciada seguindo os padrões definidos pelo esquema.

3.1.2 Arquitetura do perfSONAR

O perfSONAR é estruturado sobre uma arquitetura baseada em serviços, ou seja, cada uma de suas funções é provida por um módulo/aplicação específico para aquilo, permitindo uma separação de tarefas, bem definidas.

Cada serviço se comunica uns com os outros por meio de webservices (SOAP), englobando desde o controle de testes até a troca de informações de medições. Essa comunicação é toda feita com o uso de mensagens formatadas por esquemas NMWG, como retratado na Figura 12. Nela, pode-se observar o corpo da mensagem NMWG todo encapsulado em um pacote IP, encapsulando todas as camadas do modelo TCP/IP, acima dela, vide o a mensagem encapsulada pelo SOAP, que por sua vez é transmitido pelo HTTP, encapsulado sobre o TCP.

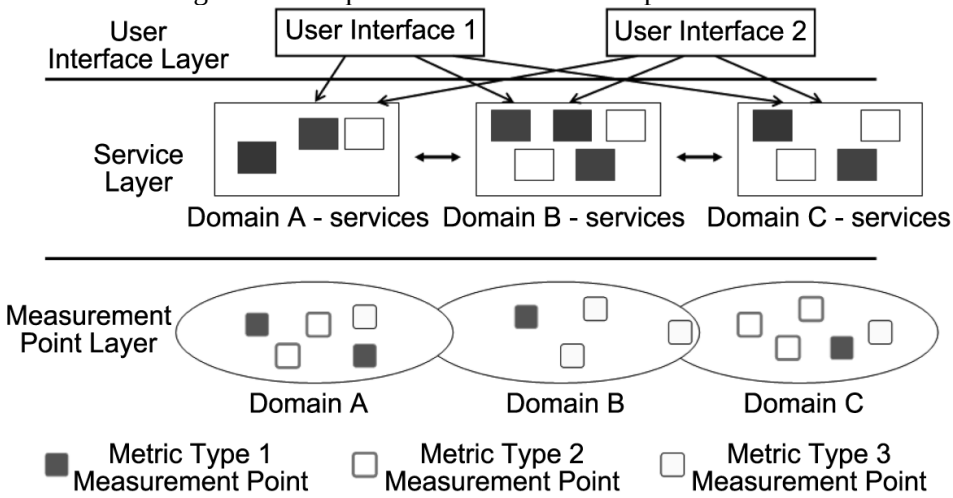
Figura 12 - Mensagem perfSONAR



Fonte: Internet2 (2014).

De acordo com o Internet2 0, a arquitetura do perfSONAR pode ser dividida em camadas (retratadas na Figura 13), que apesar de conviverem independentemente umas das outras, são capazes de se inter-relacionar de forma natural. Na figura, é possível observar três camadas macro:

Figura 13 - Arquitetura em Camadas do perfSONAR



Fonte: Internet2 (2014).

- *User Interface Layer*: interface de usuário é a camada das aplicações, na qual os dados são requisitados/retornados para o usuário. Pode-se dizer, a grosso modo, que é apenas um tradutor do código NMWG para um meio mais amigável.
- *Service Layer*: camada onde se encontram os serviços de comunicação e gerenciamento, responsáveis por fazer a intermediação entre o usuário e as medições propriamente ditas, como por exemplo o *Measurement Archive* (MA) serviço responsável por armazenar medições e as entregar, quando requisitado.
- *Measurement Point Layer*: camada onde se encontram ferramentas capazes de obter medições passivas ou ativas. Essas ferramentas podem fazer parte da solução/framework perfSONAR, ou serem terceiras. Alguns exemplos de ferramentas de medição ativa, já integradas ao framework são: IPERF, bwctl, owamp, entre outras.

3.1.3 Principais serviços do perfSONAR

O perfSONAR é capaz de automatizar o monitoramento entre redes, desde LAN a redes WAN, podendo, por exemplo, exibir a largura de banda de todos os links de um

determinado *path*[12]. Para tanto, o perfSONAR, como já mencionado, provê uma série de serviços capazes de realizar o monitoramento tanto passivo, quanto ativo. Os principais deles, segundo o Internet2 (2014), são listados abaixo:

- *Measurement Point (MP) Service*: cria e/ou publica informações de monitoramento de medições ativas ou passivas. Caso as medições estejam armazenadas, ele simplesmente as entrega ao requisitor, do contrário ele gera a informação de medição a partir de testes realizados online. O MP não é uma ferramenta propriamente dita. Conceitualmente ele se aproxima mais de um hospedeiro de aplicações voltadas para a medição, sendo normalmente um hospedeiro;
- *Measurement Archive (MA) Service*: serviço utilizado para manter um histórico das medições (mesmo que de um período recente), pois ele as armazena, possibilitando consultas posteriores, além de ser o responsável por publicar as medições que forem solicitadas, por meio de webservices;
- *Lookup Service (LS)*: funciona como uma espécie de agregador de recursos, registrando todos os serviços existentes e alcançáveis e suas respectivas capacidades de medição;
- *Authentication Service (AS)*: gerencia os níveis de acesso aos serviços, provendo o credenciamento, a fim de controlar permissões de acesso aos serviços disponíveis;
- *Topology Service (TopS)*: fornece informações de topologia das redes;

Além dos serviços supracitados, mais comumente utilziados, existem outros serviços derivados, ou que complementam o conjunto de ferramentas e serviços do perfSONAR. O OML MDIP se encaixa nesta categoria, pois será um novo serviço que complementará o leque existente no perfSONAR.

3.2 OML – ORBIT MEASUREMENT LIBRARY

Como citado anteriormente, há soluções de monitoração/medição capazes de obter dados arbitrários, ou seja, que são definidos por experimentadores de forma a atender suas necessidades, para aquele experimento, demonstrando grande flexibilidade para seus usuários.

[12] *Path* é o caminho de rede desde a origem do pacote, até o seu destino.

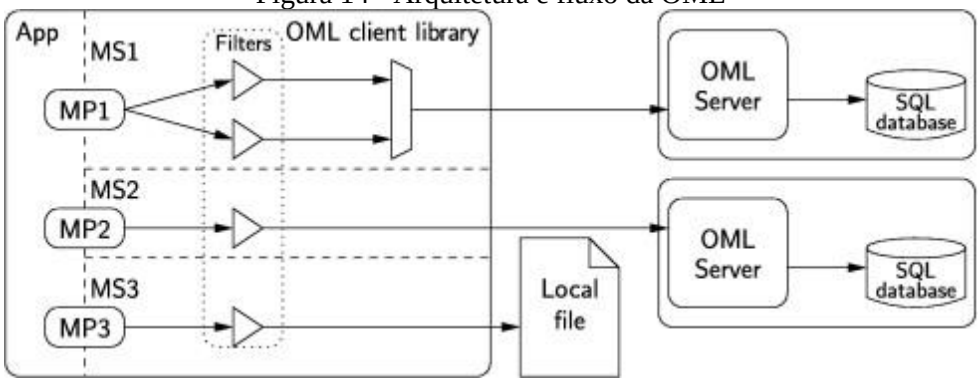
De acordo com Mehani et al (2014), a biblioteca OML é um framework baseado em uma arquitetura genérica, capaz de prover coleta e armazenamento de dados em uma base SQL. Elaborado sobre uma arquitetura cliente/servidor, ele define estruturas de dados e funções para enviar/receber e armazenar dados de medição de um experimento. A OML possui uma API, a liboml2, compatível com linguagens de programação variadas (C, Ruby, entre outras) que se integra com as aplicações utilizadas nos experimentos, como relatado em Singh et al (2005) do ponto de vista do experimentador, a OML é uma biblioteca que permite a realização de coletas de dados variados provenientes de qualquer aplicação rodando em um experimento, todas armazenadas num banco de dados remoto.

A OML foi inicialmente desenvolvido como componente do OMF, com a proposta de ser uma ferramenta de coleta de dados de medição, realizados em experimentos instrumentados pelo OMF. Atualmente, ele possui flexibilidade suficiente para se acoplar a qualquer aplicação e coletar dados de quaisquer experimentos.

3.2.1 Arquitetura da OML

Como concebido desde as suas primeiras versões Singh et al (2005), a OML é baseado em *Measurement Points* (MP), que geram *Measurement Streams* (MS) e podem ou não ser filtrados por *Filters*[13], sendo, por fim, armazenados em bancos de dados SQL ou em arquivos temporários para posterior armazenamento em banco, como bem retratado pela Figura 14. Nesta, é possível observar dois grupos de componentes: os utilizados pelos clientes (*OML Client Library*) e os utilizados pelos servidores (*OML Server*).

Figura 14 - Arquitetura e fluxo da OML



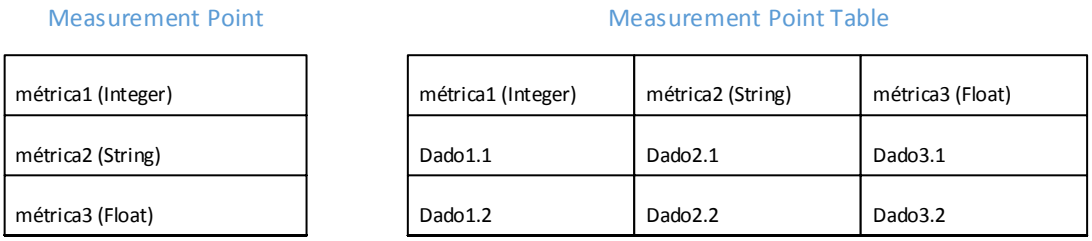
Fonte: Mehani et al (2014).

[13] *Filters* são funções do OML utilizadas para filtrar um conjunto grande de dados e/ou para manipular os dados antes de serem persistidos no banco de dados

No lado dos clientes, o primeiro e talvez o mais importante item, obrigatório para o uso da OML, é o MP. Os *Measurement Points*, estruturas de dados que definem uma coleção de métricas a serem coletadas num experimento, são definidos através de métodos existentes na **OML *Client Library*** (biblioteca de desenvolvimento da OML), no código da aplicação que está sendo utilizada. Cada MP representa uma tabela no banco de dados da OML, tendo como colunas as métricas definidas pelo experimentador. Cada uma das colunas das tabelas, poderá ser de diferentes tipos, já que cada métrica definida possui um tipo de dados (inteiro, string, etc). Enquanto isso, cada medição obtida, por consequência, se equivale a uma tupla do banco. A

Figura 15 retrata bem a estrutura citada, na qual, do lado esquerdo observa-se uma abstração de MP, tendo do seu lado direito a tabela de banco de dados representada.

Figura 15 - Representação de MP e sua tabela no ambiente OML



Fonte: Autoria Própria (2014).

Quando valores são obtidos e adicionados a seus respectivos MPs, eles são serializados e enviados ao banco de dados formando *measurement streams.* De acordo com Mehani 0, eles são caracterizados como um conjunto de métricas e seus valores, definidos em uma estrutura de dados.

Antes de serem enviados ao banco, os MPs podem ser tratados com o uso de filtros (*Filters*), que são métodos utilizados para a manipulação dos dados das medições realizadas. Apesar de não serem obrigatórios, são muito úteis quando há uma massa muito grande de medições coletadas, possibilitando o tratamento desses dados, como obtenção da média, dentre outros, facilitando a posterior visualização.

Da aplicação para o lado servidor, os MSs trafegam na rede até serem armazenados no OML *Server*, que é composto basicamente por um banco de dados SQL (SQLite ou postgreSQL). Na OML, cada experimento realizado gera uma base de dados única. Esta, terá sua estrutura (modelo de dados) definida pelos MPs, como mencionado anteriormente. Além das métricas definidas pelo experimentador, a OML define em cada tabela algumas métricas padrão, como, por exemplo, os campos PID (referindo-se ao ID do processo) e o campo *Connection ID* (que faz referência ao número identificador da conexão).

3.2.1.1 A liboml2

A liboml2 é o principal componente da OML *Client Library*, pois é o recurso que provê as funcionalidades da OML, propriamente ditas. Ela é dotada de 12 funções, todas indicadas na

Figura 16.

Figura 16 - Funções da liboml2

```
int    omlc_init(const char *appname, int *pargc, const char **argv, o_log_fn log_fn);
OmlMP* omlc_add_mp(const char *name, OmlMPDef *definition);
int    omlc_start(void);
void   omlc_inject(OmlMP *mp, OmlValueU *values);
int    omlc_close(void);
void   omlc_set_int32(OmlValueU value, int32_t int32val);
void   omlc_set_uint32(OmlValueU value, uint32_t uint32val);
void   omlc_set_int64(OmlValueU value, int64_t int64val);
void   omlc_set_uint64(OmlValueU value, uint64_t uint64val);
void   omlc_set_double(OmlValueU value, double doubleval);
void   omlc_set_string(OmlValueU value, char* stringval);
void   omlc_set_const_string(OmlValueU value, const char* stringval);
```

Fonte: Nicta (2014).

A funçãa OML**c_init** precisa ser acionada antes de qualquer outra, pois é a responsável pela inicialização da biblioteca, além de configurar a API repassando os parâmetros recebidos em sua chamada. Os parâmetros que ela aceita definem o nome da aplicação, parâmetros de entrada para aplicação, além de permitir uma metodologia de *logging* diferente do padrão, caso seja do interesse do experimentador. Quando os parâmetros informados não realizam nenhuma configuração ou indicam algum arquivo de configuração, a liboml2 provê um conjunto de filtros padrão para cada MP especificado, segundo relatado em Nicta (2014). No exemplo da

Figura 17, caso a aplicação não especifique nenhuma configuração, a biblioteca criará 2 tipos filtros padrão: o primeiro será a média (tipo avg), sendo correspondente a cada um dos itens numéricos existentes na definição do MP. O segundo, será um filtro do tipo *first*,

correspondente somente à variável do tipo string; este filtro armazena somente o primeiro valor inserido na variável.

A função OML**c_add_mp** é responsável pelo registro dos pontos de medição da aplicação. Seus parâmetros indicam o nome do MP e a estrutura do MP. A

Figura 17 exemplifica a definição de um MP, na qual é possível verificar a existência de 5 características deste ponto de medição. Cada MP é uma tupla, que só pode ser definido através dos tipos básicos *Integer, Double e String*. As versões futuras da OML darão suporte a outros tipos, além dos básicos. A definição de um MP é representada por um *array* do tipo OMLMPDef e sempre se encerra com o NULL como último elemento.

Figura 17 – Exemplo de definição de um MP

```
OmlMPDef mp_def [] =
{
  { "source", OML_UINT32_VALUE },
  { "destination", OML_UINT32_VALUE },
  { "length", OML_UINT32_VALUE },
  { "weight", OML_DOUBLE_VALUE },
  { "protocol", OML_STRING_VALUE },
  { NULL, (OmlValueT)0 } /* Sentinel value */
};
```

Fonte: Nicta (2014).

A **omlc_start** é a inicializadora de todo o processo, de fato. Obrigatoriamente deverá estar após as funções supracitadas.

Com a funçãa OML**c_inject** é possível atribuir valores aos MPs. Para tal, além de registrar os MPs, é necessário criar *arrays* do tipa OMLValueU, de mesmo tamanho da quantidade de características do MP. Por exemplo, na

Figura 17, para o MP definido, será necessário criar um *array* OmlValueU de tamanho 5.

A função para finalizar a biblioteca, ou seja, finalizar a coleta de medições, é a **omlc_close**.

As funções **oml_set_** servem para atribuir valores aos *arrays* do tipa OMLValueU, utilizado na função de atribuição dos MPs (omlc_inject).

A Figura 18 retrata o que seria uma aplicação (app) instrumentada com a OML. Nela observa-se o uso de métodos provenientes de uma biblioteca, a liboml2, responsável por integrar aplicações à arquitetura OML.

No exemplo da Figura 18, é possível observar todos os estados da OML, desde sua inicialização, seguida da criação do MPs, além da transformação do MP em MS, para posterior inserção no banco de dados (OML_inject), e por fim o encerramento do processo, comunicando ao sistema que não haverá novos MSs trafegando.

Figura 18 - Instrumentação de um app com a liboml

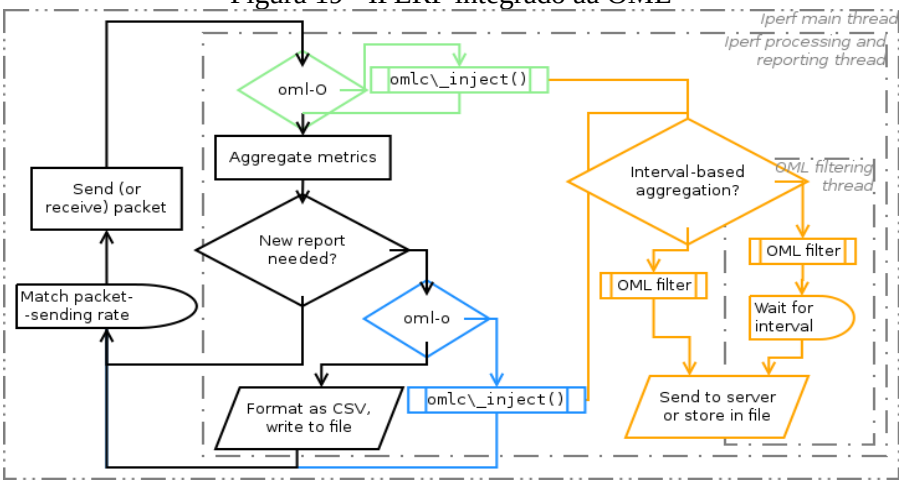
```
/* example.c */
#define OML_FROM_MAIN           /* Defining OML_FROM_MAIN enables the
    generated code in the next include */
#include "example_oml.h"        /* example_oml.h is generated by
    oml2-scaffold --oml example.rb */

int main(int argc, const char** argv) {
  char *label; int n;

  omlc_init("example", &argc, argv, NULL); /* Initialise OML according to
      the command line arguments*/
  oml_register_mps();           /* Register the MPs to OML so their schema
      can be put in the MS, and injection of metadata */
  omlc_start();                 /* Establish a connection to a sink, and
      send headers */


  while(1) {
    application_measure_something(label, &n);
    oml_inject_example_mp(g_oml_mps->example_mp, label, n); /* Inject a new
        tuple into the MS; helper generated by oml2-scaffold(1) */
    do_some_other_things();
  }

  omlc_close();                 /* We're done */
}
```

Fonte: Mehani et al (2014).

3.2.2 Exemplo de uso da OML

Vários são os possíveis casos no qual a OML se encaixa e pode ser posto como a ferramenta de coleta para os diversos dados de medição. Um exemplo muito utilizado e eficaz, que também foi adotado neste trabalho, é a realização de experimentos para verificação de perda e largura de banda da rede, fazendo uso do IPERF. A Figura 19 demonstra o fluxo de como seria o IPERF integrado aa OML. Nela é possível observar o fluxo do IPERF (linhas em preto) e os momentos de interação com a OML (linhas azuis), principalmente a chamada ao método OML_inject e a chamada às funções de filtragem, finalizando com a inserção no OML *Server*.

Figura 19 - IPERF integrado aa OML



Fonte: Nicta (2014).

Na aplicação em questão, foram definidos sete MPs (*Application*, *Connection*, *Settings*, *Transfer*, *Losses*, *Jitter*, *Packets*) a fim de deixar o experimento pronto para qualquer que seja o objetivo do experimentador (medir a taxa de perda, largura de banda, atraso, etc), flexibilizando o uso da aplicação.

O experimento para este exemplo, objetivou medir a taxa de transferência entre dois hospedeiros. O hospedeiroA, um computador físico localizado no laboratório GROW da UNIFACS, preparado para rodar exclusivamente este experimento. A outra ponta foi o hospedeiroB, servidor instalado nas dependências da UFPE, rodando o perfSONAR BUOY[14]. Os testes foram executados no turno da noite, após as 22h, horário de pouco uso dos links dessas instituições, com fins exclusivos de demonstração de funcionalidade.

Quadro 3 - Definição de MPs (Parâmetros DE App à esquerda – 3ª. Parâmetros de Conexão à direita – 3B)

Name	Type	Description
Pid	integer	Main process identifier
Version	string	IPERF version
Cmdline	string	IPERF invocation command line
starttime_s	integer	Time the application was received (s)
starttime_us	integer	Time the application was received (us)

Name	Type	Description
pid	integer	Main process identifier
connection_id	integer	Connection identifier (socket)
local_address	string	Local network address
local_port	integer	Local port
remote_address	string	Remote network address
remote_port	integer	Remote port

Fonte: Nicta (2014).

Os Quadro 3 e Quadro 4 retratam os *Measurement Points* a nível de bancos de dados, ou seja, o formato como eles serão exibidos na base de dados do OML *Server*.

Quadro 4 - Definição de MPs (Configurações de App à esquerda – 4A e Dados de transferência à direita – 4B)

Name	Type	Description
pid	integer	Main process identifier
server_mode	integer	1 if in server mode, 0 otherwise
bind_address	string	Address to bind
multicast	integer	1 if listening to a Multicast group
multicast_ttl	integer	Multicast TTL if relevant
transport_protocol	integer	Transport protocol (IANA number)
window_size	integer	TCP window size
buffer_size	integer	UDP buffer size

Name	Type	Description
Pid	integer	Main process identifier
connection_id	integer	Connection identifier (socket)
begin_interval	float	Start of the averaging interval (IPERF timestamp)
end_interval	float	End of the averaging interval (IPERF timestamp)
Size	uint64	Amount of transmitted data [Bytes]

Fonte: Nicta (2014).

As informações dos quadros acima foram extraídas a partir de um dump da base gerada após a realização do experimento. Antes da realização, como já citado, os MPs devem ser definidos dentro da aplicação. A Figura 20 ilustra como o MP referente ao quadro 4B foi definido dentro da aplicação utilizada no experimento.

[14] O perfSONAR BUOY é um módulo do perfSONAR com funções de MA e testes, através do BWTCL e do OWAMP

Figura 20 – Definição de MP na app do IPERF

```
static OmlMPDef oml_transfer_def[] = {
  {"pid", OML_INT32_VALUE},
  {"connection_id", OML_INT32_VALUE},
  {"begin_interval", OML_DOUBLE_VALUE},
  {"end_interval", OML_DOUBLE_VALUE},
  {"size", OML_UINT64_VALUE},
  {NULL, (OmlValueT)0}
};
```

Fonte: Nicta (2014).

Para a realização desse experimento, o hospedeiroB foi escolhido como cliente, deixando a função de servidor para o hospedeiroA, que precisou ter o IPERF rodando inicialmente nele, através do comando "IPERF –s", sendo o "-s" o indicativo que o IPERF irá rodar no modo servidor. No hospedeiroA, o comando executado utilizava somente o parâmetro indicando quem seria o servidor, deixando o comando como segue: "IPERF –c ip_do_servidor". Já que o objetivo dos testes era de apenas conhecer a largura de banda disponível entre os hospedeiros, naquele momento, o IPERF foi executado em modo padrão, sem parâmetros diferentes do exigido, como mencionado anteriormente.

Apesar da aplicação do IPERF integrado aa OML estar preparada para obter todos os resultados de medições que o IPERF é capaz de entregar, este caso de uso se limitou à largura de banda e isso fez com que apenas 4 MPs obtivessem valor durante o experimento: *Application*, *Connection*, *Settings*, *Transfer*. O Quadro 5, observa-se o conteúdo obtido com a realização do experimento.

Quadro 5 - Tabela Transfer (resultado de um experimento)

pid	connection id	begin interval	end interval	size
12758	4	0	1.000.000	1212153856
12758	4	1.000.000	2.000.000	1219887104
12758	4	2.000.000	3.000.000	1239810048
12758	4	3.000.000	4.000.000	1229979648
12758	4	4.000.000	5.000.000	1236795392
12758	4	5.000.000	6.000.000	1246232576
12758	4	6.000.000	7.000.000	1241776128
12758	4	7.000.000	8.000.000	1240465408
12758	4	8.000.000	9.000.000	1232207872
12758	4	9.000.000	10.000.000	1199046656

Fonte: Autoria Própria (2014).

Analisando o Quadro 5, é possível observar vários fatores interessantes, visto que são extremamente esclarecedores, a exemplo do campo "size" que indica a quantidade de dados transferidos (em bytes). Esses dados se apresentam no resultado como praticamente

constantes em todos os intervalos, demonstrando que a banda disponível no momento dos testes era de algo próximo a 1 Gbps.

Dadas os esclarecimentos sobre como todas as tecnologias utilizadas para a construção do MDIP funcionam, os capítulos posteriores focarão em demonstrar como o MDIP foi contruíso e como se dá seu funcionamento.

4 OML MDIP

O OML MDIP é uma solução compatível com o perfSONAR, concebido com a finalidade de convergir os dados de medição de redes para experimentação baseadas no OMF, para um ambiente mais uniformizado e homogêneo, vislumbrando a obtenção dos dados, de forma simplificada.

O OML MDIP teve sua ideia original desenvolvida a partir de 0, quando da proposição da FBRE-BR I&M *Architecture* (*Instrumentation and Measurement Architecture*), no contexto do *testbed* FIBRE. A solução é focada numa metodologia para exportar os dados de experimentos realizados com o uso do OMF, para o ambiente deste *testbed*, entregando todos os resultados por meio de uma solução capaz de entregar os dados de medição de forma simples e unificada.

Desenvolvido sobre a *engine* do perfSONAR SQLMA (SQL *Measurement Archive*) o OML MDIP tem como essência, a mesma de qualquer outro MA do perfSONAR, ou seja, disponibilizar os dados de medições armazenados em uma base SQL, ou seja, especificamente o MDIP desenvolvido possui a capacidade de acessar bancos de dados do OML *Server*, desde que possuam uma tabela denominada FibreMP, que torna o banco compatível com o OML MDIP.

Vislumbrando um nível de acesso dos dados de experimentos completamente independente da ilha onde o experimento foi realizado, o OML MDIP faz uso do iRODS para armazenar os dados, possibilitando que o experimentador possa ter acesso a eles independente de localização geográfica, além de garantir um backup dos experimentos realizados, como proposto por Hohlenweger 0.

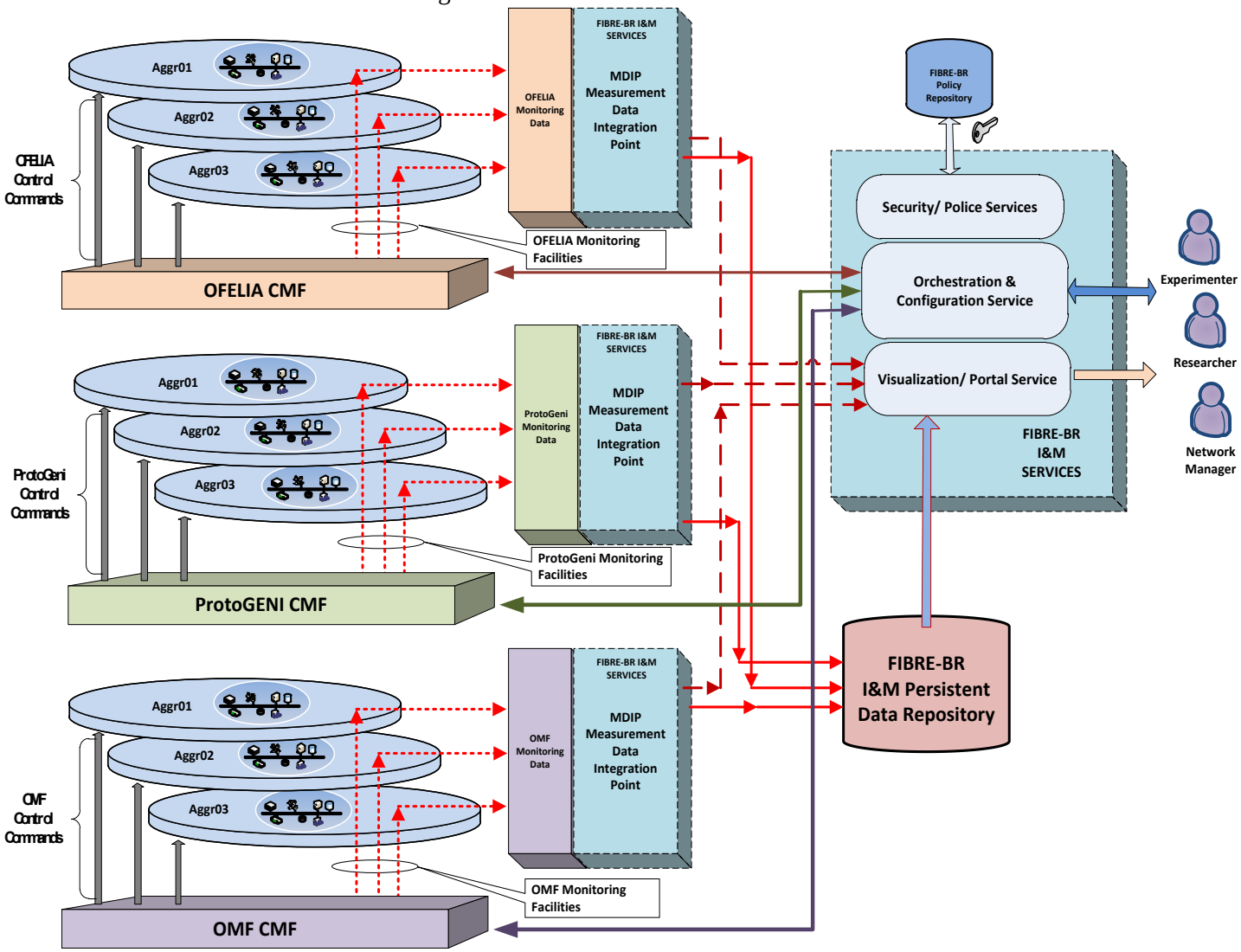
4.1 FIBRE-BR I&M *ARCHITECTURE*

Em 2012, o grupo de monitoramento da UNIFACS e UFPE propôs a *FIBRE–BR* I&M *Architecture* (Arquitetura de medição e instrumentação do FIBRE-BR), que se apresenta como uma solução capaz de configurar, monitorar, coletar e exibir os dados da infraestrutura e dados específicos de experimentos de agregados de CMFs.

A Figura 21 ilustra a proposição feita pelo time e apresentada em Pinheiro (2012), criada com inspiração nas capacidades já conhecidas dos CMFs. Nela, é possível observar abstrações dos CMFs OFELIA, Protogeni e OMF, capazes de se comunicar com o módulo de

I&M *Services* por meio de MDIPs – *Measurement Data Integration Point* (Ponto de integração e de dados de medição) – e armazenar seus dados no I&M *Persistent Data Repository* (Repositório de Dados persistentes do I&M).

Figura 21 – FIBRE-BR I&M Architecture



Fonte: Pinheiro et.al (2012).

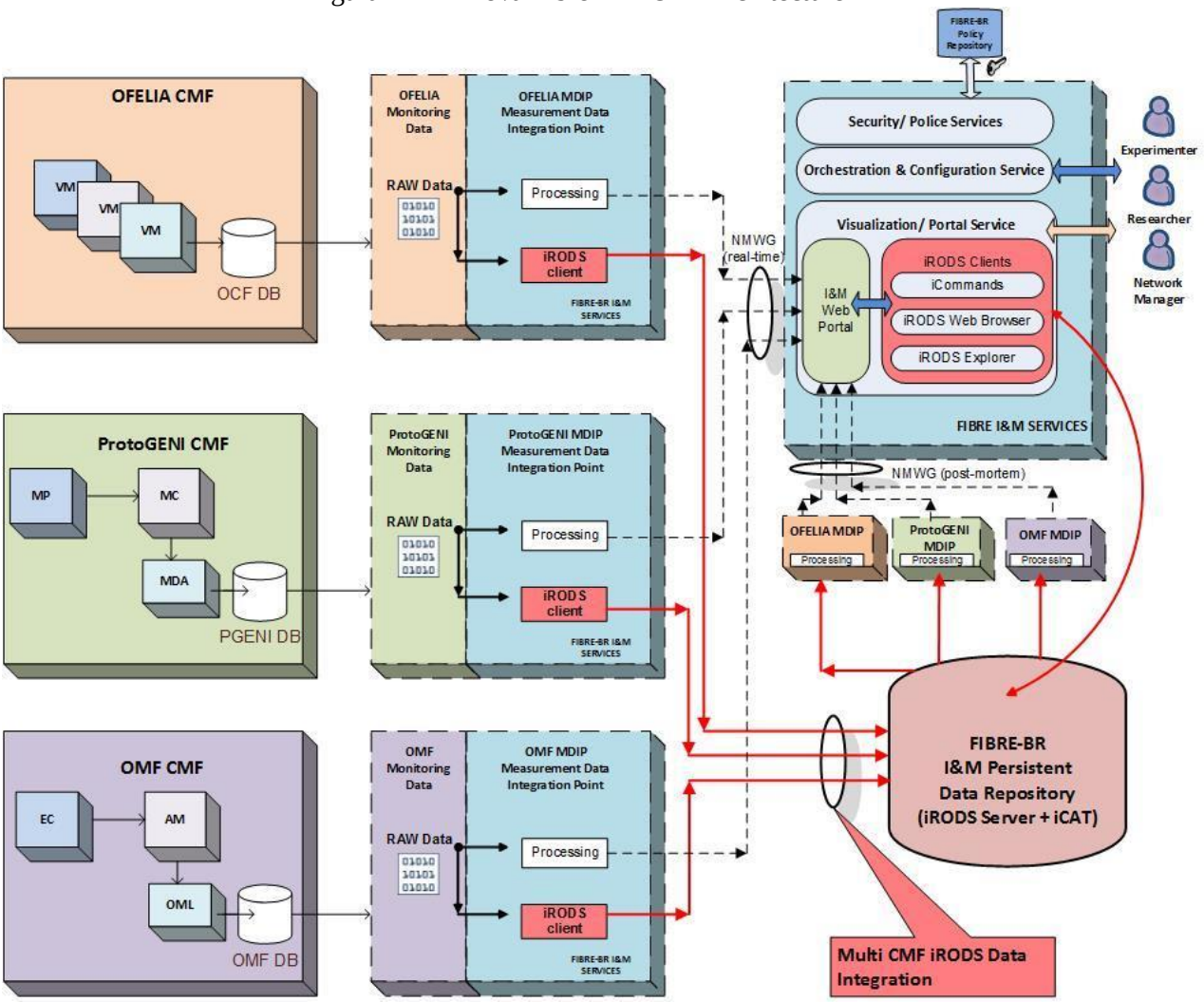
O módulo *Orchestration and Configuration Services* (Serviços de Orquestração e Configuração) objetiva configurar e definir pontos de medição[15], além de orquestrar os experimentos realizados nos múltiplos CMFs. Já o *I&M Portal* (Portal do I&M) deve prover uma interface que facilite o acesso aos dados de medições e resultados dos experimentos.

[15] Pontos de Medição, são hospedeiros geradores de informações de monitoramento, podendo ser passivos ou ativos no processo de monitoração.

O MDIP, módulo cuja principal função é a integração dos dados, é responsável por colocá-los em conformidade com os padrões do FIBRE I&M, ou seja, ele lida, principalmente, com o formato dos dados e questões de armazenamento.

Baseado no mesmo conceito do *Measurement Archive* – MA (Arquivador de medições) do perfSONAR, um MDIP possui, dentre outras, a capacidade de explicitar dados coletados via SNMP, banco de dados SQL ou bases *round robin* (RRD), além de prover normatização dos dados com o formato padrão NMWG, objetivando a "exportação" dos mesmos, principalmente, para o I&M *Portal*, salvaguardadas as particularidades dos CMFs, exigindo que cada MDIP tenha uma implementação própria.

Figura 22 – A nova Fibre-BR I&M Architecture



Fonte: Suruagy e Martins (2013).

O repositório persistente proposto na arquitetura (Fibre *I&M Persistent Data Repository*), previa uma solução abstrata de armazenamento, para posterior obtenção dos dados de experimentação, seguindo um conceito similar aos MAs do perfSONAR. Em 2013, Suruagy e Martins (2013) apresentaram o iRODS como solução de armazenamento, como

pode ser visto na Figura 22, deixando o conceito do MA, de coletar os dados a partir de uma base única, parcialmente de lado.

Suruagy e Martins (2013) apresentaram uma metodologia para persistência global de informações referentes ao experimento utilizando o iRODS, ou seja, ao invés de armazenar somente os dados de medição, a proposta possibilitou que tudo referente ao experimento, como por exemplo, o "arquivo de instrumentação"[16] do experimento, arquivos de máquinas virtuais, arquivos de bancos de dados, entre outros, pudessem ser armazenados no iRODS.

Além do backup de qualquer informação, em qualquer formato, o iRODS funciona de maneira distribuída, permitindo que a informação armazenada nessa base, seja acessada de forma transparente para o usuário, como se o dado estivesse num servidor local.

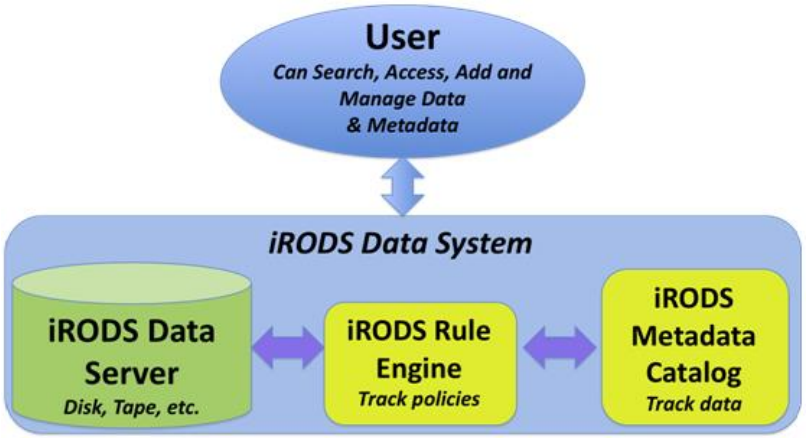
4.2 iRODS

O OML MDIP, além de acessar bases de dados de um OML *Server*, possibilitando a entrega de dados solicitados, de forma uniformizada, permite que dados de experimentos realizados em *testbeds* (redes para experimentação) diferentes possam ser acessados a partir do *testbed* local, de forma transparente, fazendo uso do iRODS.

O iRODS é um sistema de *data grid* utilizado para consolidação de sistemas de dados (IRODS WIKI, 2014). É capaz de prover maneiras de gerenciar, organizar, compartilhar, proteger e preservar dados de e entre usuários, por meio de uma estrutura hierárquica, possibilitando a construção de uma solução de armazenamento (centralizada ou distribuída) se assemelhando a um diretório de arquivos comum.

[16] Para o OMF, por exemplo, um arquivo de instrumentação é um arquivo que contém todas as instruções e reservas de quais recursos serão utilizados no experimento e como eles serão configurados.

Figura 23 – Componentes do iRODS



Fonte: TWGRID (2014).

Como mostrado na Figura 23, o iRODS é composto basicamente por duas funções principais: a de usuário e o seu sistema propriamente dito (*iRODS Data System*). O primeiro, representado na Figura 23 como "*User*", é composto por um conjunto de ferramentas capazes de consultar, adicionar, remover, entre outros, os dados na base do iRODS. Esse conjunto é denominado ***iRODS iClients***, e pode ser instalado em qualquer hospedeiro que possua conectividade com o *iRODS Data System.*

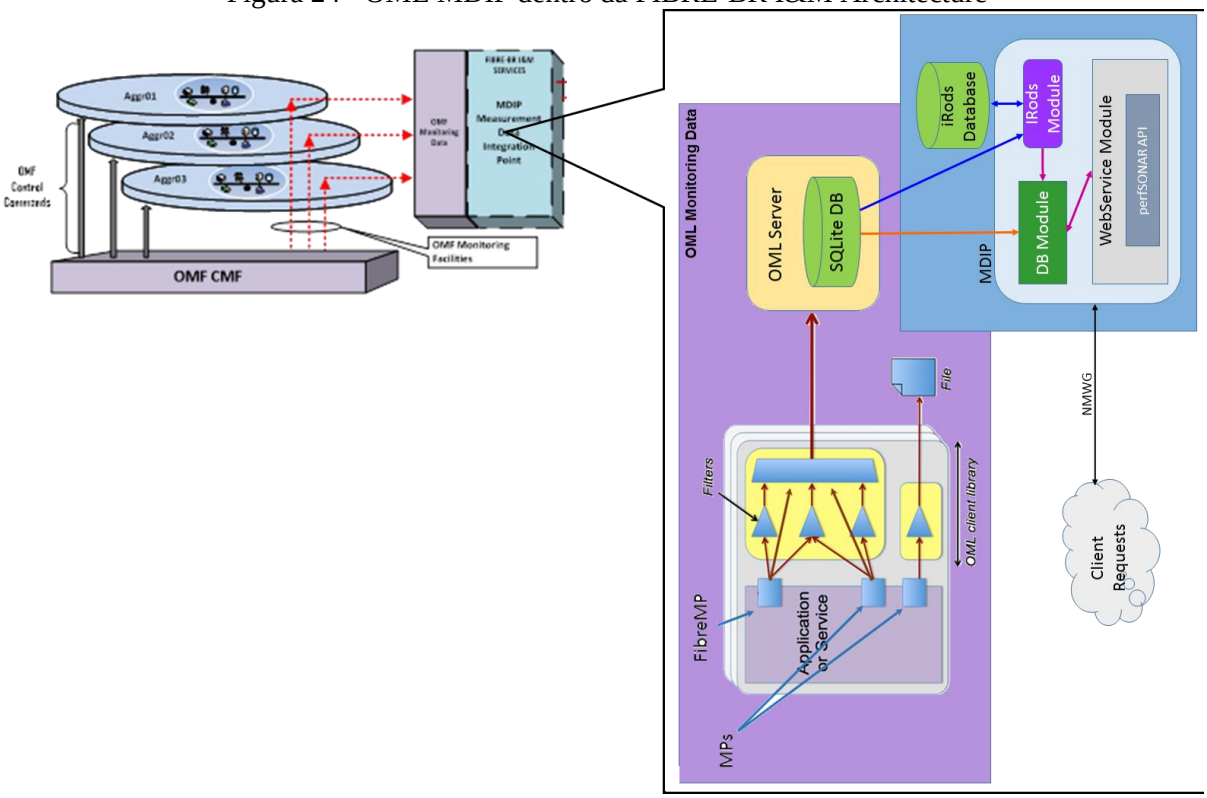
A segunda função do iRODS, o ***iRODS Data System*** é composto por três módulos, que funcionam de forma interdependente. O ***iRODS Data Server*** é responsável por interagir com mídia de armazenamento (fita, disco, nuvem, etc), possibilitando além do armazenamento propriamente, o compartilhamento de arquivo ou pastas. O ***iRODS Rule Engine*** faz o controle de políticas para os arquivos armazenados, incluindo controle de acesso, níveis de apresentação, replicação e distribuição, dentre outros. Por último, o ***iRODS, Metadata Catalog*** ou **iCAT** é o módulo responsável por catalogar e descrever toda a informação de metadado, ou seja, todas as informações que descrevem os arquivos armazenados, contribuindo de forma significativa para o bom desempenho do sistema.

4.3 ARQUITETURA DO OML MDIP

O OML MDIP foi concebido a partir da necessidade da arquitetura I&M em integrar o OMF, seu foco principal, ao conjunto de módulos do FIBRE-BR I&M *Architecture*.

A Figura 24 ilustra o que vem a ser o OML MDIP dentro da representação do mesmo no FIBRE-BR I&M Architecture. Cada um dos componentes do MDIP, serão explicados com detalhes, posteriormente neste capítulo.

Figura 24 - OML MDIP dentro da FIBRE-BR I&M Architecture



Fonte: Autoria Própria (2014).

O OML MDIP vem a ser uma derivação do perfSONAR MA, e especificamente neste caso, ele é derivado do perfSONAR SQLMA (GEANT, 2014), um MA capaz de coletar dados bem definidos e estruturados, armazenados em um banco de dados MySQL, e entregá-los ao requisitante que executou todas as solicitações fazendo uso dos padrões e protocolos definidos para tal.

Apesar de seu objetivo em comum com outros MAs, o OML MDIP diferencia-se principalmente no fato de que a base de dados da qual ele obtém as informações solicitadas, é gerada por uma ferramenta completamente independente de sua arquitetura, podendo ser uma base compatível (a partir da qual ele obtém os dados) ou não compatível (não sendo possível realizar consultas nessa base)

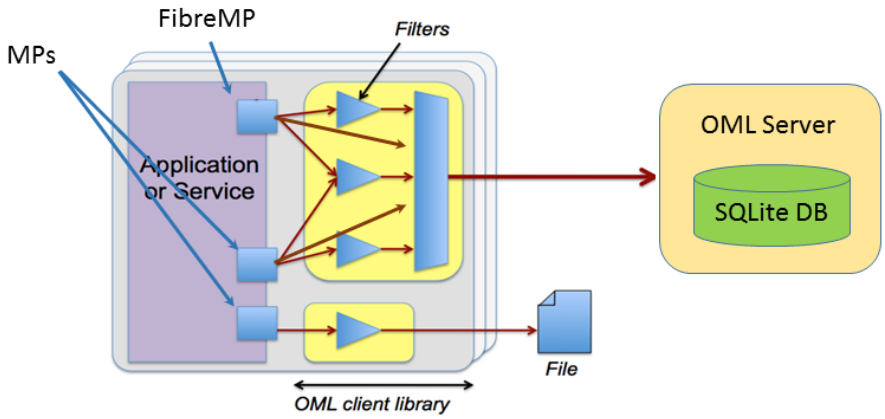
A principal função do OML MDIP é exportar dados de medição, armazenados em bancos de dados do OML Server, de forma padronizada, permitindo aos usuários a integração das bases do *OML Server* com outros ambientes, visto que os dados solicitados serão entregues num formato conhecido. Isso por si só, é considerada a principal diferença do OML MDIP para a OML, visto que enquanto o pimeiro necessita de alguma formalização de como os dados são aramazenados no banco, o segundo permite que o usuário experimentador defina

as métricas a serem coletadas da forma que ele considerar mais adequada, podendo definir as métricas como achar mais pertinente.

Visando tratar a questão da extrema flexibilidade da OML, no que se refere à definição de *measurement points*[17], que no OML *Server* são tabelas de bancos de dados, o OML MDIP propõe que todos os experimentos que utilizem a OML como coletor das medições, definam no mínimo um MP, denominado "**FibreMP**", que possibilitará a uniformização dos dados de medição.

O "FibreMP", é um *Measurement Point* diferenciado, visto que ele já pré-define (normatiza), contextualizando uma série de métricas, conforme documentado no Apêndice A.1, a fim de entregar ao experimentador uma vasta gama de variáveis para coleta de informações de medição, de forma organizada, padronizando a maneira como o MDIP entregará os dados aos solicitantes.

Figura 25 - Arquitetura da OML



Fonte: Nicta (2014).

A Figura 25 ilustra a arquitetura da OML, exemplificando o uso do FibreMP, que da mesma forma que outros MPs, podem, ou não, ser filtrados, e processados em *measurement streams*, para serem armazenados no banco de dados do OML *Server*, ou em um arquivo, quando a conectividade ao banco de dados não for possível.

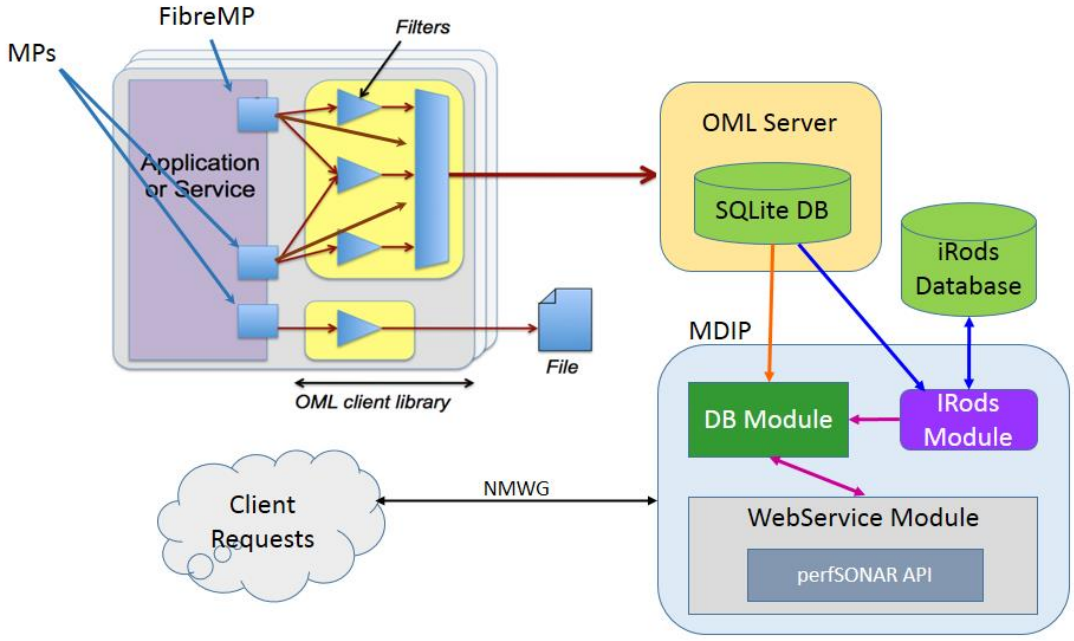
4.3.1 Componentes do OML MDIP

O OML MDIP é formado por um conjunto de quatro módulos (*Webservice Module, Database Module, iRODS Database e iRODS Module*), desenvolvidos utilizando a linguagem de programação JAVA. Ele utiliza a arquitetura base da OML, estendendo-a por meio de

[17] *Measurement Points*, no OML, são estruturas de dados que armazenarão os dados de medição.

módulos, possibilitando o acesso aos dados da OML sem que seja necessário acessar o banco de dados diretamente, tornando esse processo transparente ao usuário.

Figura 26 - Arquitetura do OML MDIP



Fonte: Autoria Própria (2014).

A Figura 26 ilustra toda a arquitetura do OML MDIP. Nela é possível observar a arquitetura da OML, sendo estendida pelo MDIP. É possível perceber também o fluxo de comunicação dos módulos, a exemplo do *DB Module*, que não tem capacidade de enviar dados ao OML *Server*, apenas de receber o que foi requisitado.

O ***webservice*** construído é uma extensão do mesmo componente utilizado no perfSONAR SQLMA. Ele foi modificado com a finalidade de atender as seguintes funções:

- Receber requisições compatíveis com o esquema do OML MDIP (Apêndice A.2);
- Interpretar as requisições recebidas de forma a obter os dados relevantes solicitados na query de requisição SOAP;
- Repassar os dados interpretados ao módulo de acesso ao banco de dados;
- Receber o resultado da consulta feita no banco, de forma a montar a resposta SOAP e entregá-la ao requisitante;

Para ter acesso ao banco de dados, o *webservice* se comunica com o "***DB Module***". Esse módulo é responsável por:

- Receber os parâmetros de consulta do banco, que o usuário desejar, já interpretados pelo webservice;

- Executar a consulta a fim de obter a informação solicitada na query;
- Entregar o resultado de forma "bruta" ao webservice, para que ele possa organizá-los no XML de resposta e repassá-lo ao requisitante.

Os módulos apresentados até então, por si só, já representam a principal funcionalidade do MDIP. No entanto, analisando a Figura 26, observa-se ainda mais dois componentes: *iRODS Module* e *iRODS Database*.

O ***iRODS Database***, como já mencionado, é o componente escolhido para atender a proposta apresentado por Pinheiro et. al. (2012) e avaliada por Hohlenweger (2014), que vislumbra o armazenamento dos dados de experimentação, ou como neste caso, os dados de medição.

Já o ***iRODS Module*** é o componente responsável pela integração entre as tecnologias (OML e iRods), permitindo que os BDs do OML *Server* sejam armazenados diretamente na base do iRODS. Além disso, esse módulo também permite que bases que não estejam disponíveis localmente no servidor da OML, possam ser "resgatadas" e posteriormente consultadas pelo *DB Module*.

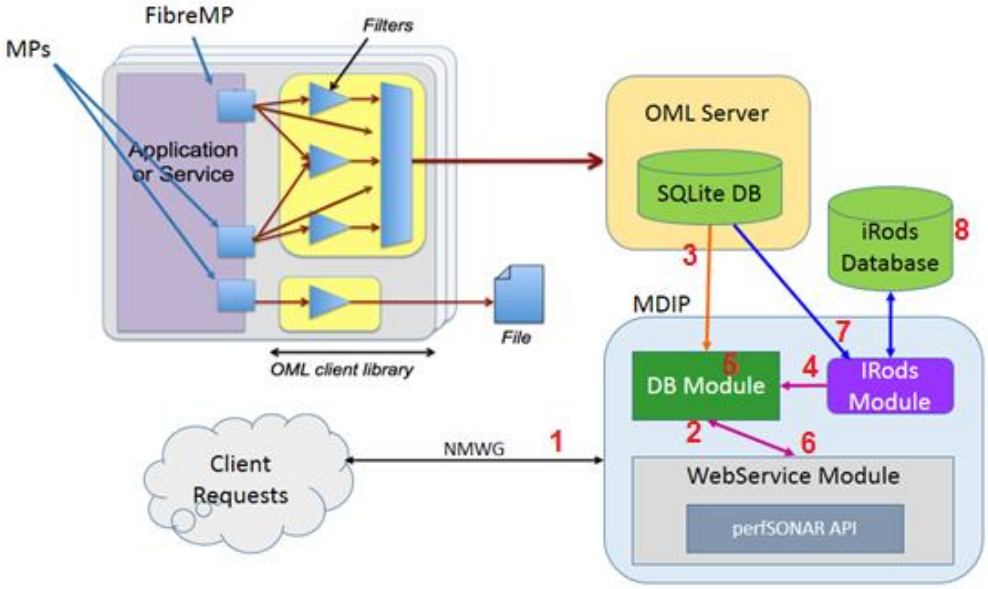
A nível de implementação, cada um desses módulos está representado por uma ou mais classes JAVA, expostas nos apêndices B.1, B.2 e B.3, deste trabalho.

4.4 FUNCIONAMENTO DO MDIP

Conforme mencionado anteriormente, o OML MDIP possui o mesmo princípio existente entre todos os MDIPs, sendo capaz de entregar os dados de medição obtidos a partir de experimentos realizados em CMFs, de forma padronizada, uniformizando a maneira que os requisitantes realizam suas consultas e recebem os resultados. O que o distingue dos demais é de onde ele coleta os dados, neste caso da base OML, além da proposta de uniformização dos dados, que a OML não possui.

Quando um experimentador/usuário utiliza o MDIP, toda a camada de acesso aos dados de experimento da OML se torna transparente, visto que o usuário deixa de acessá-lo diretamente. Por meio de uma aplicação capaz de consumir o *webservice* SOAP provido pelo MDIP, o experimentador consegue especificar os dados que precisa, a fim de obter os dados de medição gerados a partir do experimento, como resposta.

Figura 27 – Fluxo da Arquitetura do OML MDIP



Fonte: Autoria Própria (2014).

A aplicação consumidora do *webservice*, representada na Figura 27 como "*Client Requests*", deverá ser capaz de gerar uma requisição SOAP **(1)**, em conformidade com os esquemas NMWG, contendo informações como: nome do bando de dados a ser consultado (identificado pelo metadataid), parâmetros de filtragem de dados (lista de parâmetros a comporem a query do banco), dentre outros, indicados no esquema do OMLMDIP no apêndice A.2.

O *webservice module* do MDIP recebe as requisições **(2)**, as interpreta, obtendo informações como o nome do banco que será consultado, os parâmetros que serão utilizados como filtro na *query* SQL, repassando essas informações ao módulo de banco de dados, para seu devido processamento.

Com os parâmetros recebidos do webservice, o *DB Module* busca o arquivo referente à base de dados em seu repositório e abre uma conexão com o banco **(3)**. Caso o BD não exista em seu repositório, ele aciona o *iRODS Module* **(4)** para verificar se a base de dados indicada pode ser encontrada num repositório diferente e, caso isso seja possível, a entrega para o *DB Module*, que monta a *query* **(5)**, e a executa a fim de obter os resultados solicitados.

Com a etapa supracitada concluída, o resultado da consulta é repassado para o *webservice* **(6)**, que gera a resposta SOAP em conformidade com o NMWG, enviando-a para o cliente que fez a requisição inicial.

Periodicamente, o *iRODS Module* do OML MDIP acessa o repositório de bancos de dados SQLite **(7)** do OML *Server* que ele está instalado e faz um backup dos arquivos de banco existentes, com a finalidade de salvaguardar os arquivos de banco **(8)** no *iRODS*

Database, além de permitir que as bases possam ser consultadas a partir de localidades diferentes do original, desde que as bases do iRODS estejam sincronizadas.

5 VALIDAÇÃO DO OML MDIP – PROVA DE CONCEITO

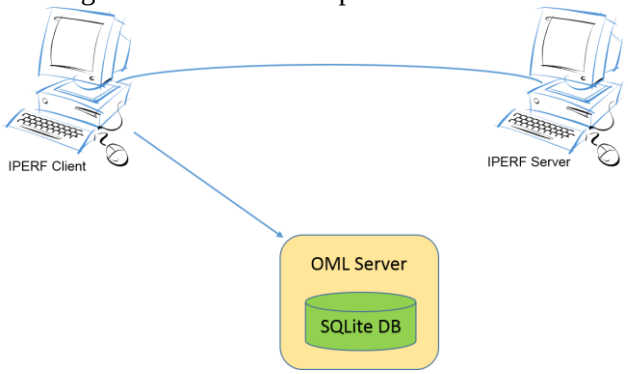
A fim de validar a aplicação desenvolvida (OML MDIP), esta seção se propõe a demonstrar um experimento, que objetiva coletar os instantes em que um pacote é enviado e recebido pelo cliente e servidor do IPERF[18]. Para tanto, o código fonte do IPERF foi modificado de forma a se tornar compatível com a OML, possibilitando que os dados medidos pudessem ser armazenados pelo OML *Server*. No decorrer deste capítulo, os detalhes da implementação dessa validação serão expostos.

5.1 PREPARAÇÃO PARA O EXPERIMENTO

Para tornar o IPERF compatível com a OML, ou seja, fazer com que o IPERF possa gravar os dados de medição na base da OML, foi necessário alterar o código fonte da app, de forma a explicitar os *Measurement Points* a serem capturados pela OML, fazendo a posterior compilação de todo o código. A versão dos códigos do IPERF, utilizados para edição, são da versão 2.0.2, todo escrito na linguagem C.

Além da modificação do código do IPERF, foi necessário criar um arquivo texto, com a extensão ".in", utilizando a linguagem para definição de experimentos do OMF, a OEDL. Esse arquivo é utilizado durante o processo de compilação do código fonte do IPERF. A Figura 28 representa uma aplicação IPERF compatível com a OML. Nela, observa-se dois hospedeiros conectados entre si, rodando o IPERF, um em modo cliente e o outro modo servidor, porém apenas o IPERF *Client* tem acesso ao OML *Server*, sendo o único a ser efetivamente compatível com a OML.

Figura 28 - IPERF compatível com a OML



Fonte: Autoria Própria (2014).

[18] De acordo com 0, IPERF é uma ferramenta para medir a largura máxima de banda disponível, permitindo o ajuste dde vários parâmetros e caracrísticas do UDP.

O arquivo ".in" definido é encontrado no Apêndice C deste trabalho. Nele, é possível verificar todas as definições necessárias para a realização do experimento dentro do ambiente OMF, inclusive a definição dos *Measurement Points.*

O exemplo mostrado no Quadro 6, traz a definição do *Measurement Point* "fibremp" que, como explicado anteriormente, é requisito necessário para tornar o experimento compatível com o perfSONAR. Nesse exemplo, faz-se uso das métricas "*connection_id*", "*begin_interval*" e "*end_interval*", para obter o identificador da seção, e os instantes do envio e do recebimento do pacote, respectivamente. As métricas definidas como *Measurement Points* serão todas correspondentes a campos de uma tabela de banco de dados, armazenado no OML *Server*.

Além das métricas supracitadas, a métrica "*size*" é utilizada para obter o tamanho do pacote trafegado. A métrica "pid" é utilizada em todas as definições do *measurement points*, e serve para identificar o processo gerador dos dados, no entanto, ela não é relevante para o processo de validação.

Quadro 6 - Definição do FibreMP dentro do arquivo ".in"

```
a.defMeasurement("fibremp"){ |m|
   m.defMetric('pid', :integer, 'Main process identifier')
   m.defMetric('connection_id', :integer, 'Connection identifier (socket)')
   m.defMetric('begin_interval', :float, 'Start of the averaging interval (IPERF timestamp)')
   m.defMetric('end_interval', :float, 'End of the averaging interval (IPERF timestamp)')
   m.defMetric('size', :uint64, 'Amount of transmitted data [Bytes]')
 }
```

Fonte: Autoria Própria (2014).

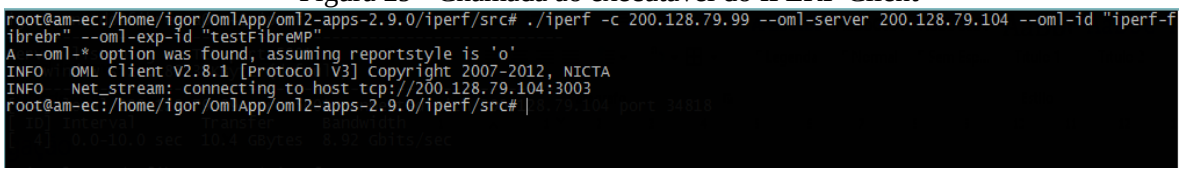
Com o código fonte da aplicação modificado, já com o MPs definidos e toda a definição do experimento detalhada no arquivo ".in", é necessário preparar a aplicação para ser compilada e depois compilar os códigos, a fim de se obter o executável da aplicação. Para tanto, utiliza-se os comandos "./configure" e "make", repectivamente. Dessa forma, a aplicação está pronta para ser executada, permitindo que o experimento seja realizado, conforme as diretrizes do experimentador.

5.2 EXECUÇÃO DO EXPERIMENTO

Com a aplicação pronta para ser executada, na linha de comando, fez-se uma chamada ao executável dela, conforme representado pela Figura 29, indicando como parâmetros:

- oml-server: utilizado para indicar o servidor da OML, onde os dados serão armazenados. Durante o experimento, o servidor da OML utilizado foi o 200.128.79.104;
- oml-id: identificador da seção aberta por aquele processo. Na execução deste experimento, utilizou-se o identificador "IPERF-fibrebr";
- oml-exp-id: identificador do experimento, que também será utilizado como nome da base criada, que neste caso é "testFibreMP". Se não explicitado, a OML-exp-id é gerado baseado na hora, minuto, segundo e milissegundos do momento da execução do comando;

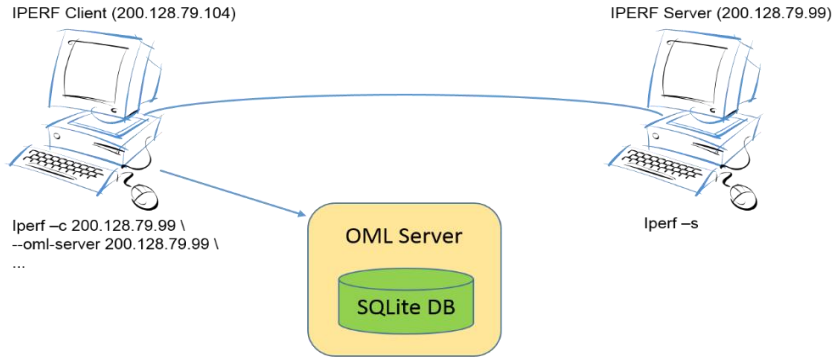
Figura 29 - Chamada ao executável do IPERF Client

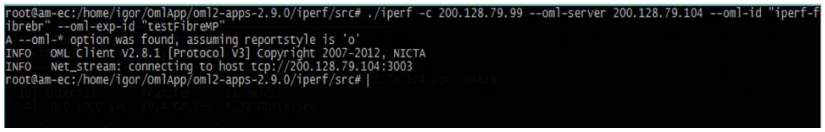


Fonte: Autoria Própria (2014).

A Figura 30, além de trazer novamente a chamada ao executável do IPERF, traz uma representação gráfica da execução da aplicação, na qual de um lado o IPERF é executado em modo cliente (característica evidenciada pelo parâmetro "-c") buscando o servidor do IPERF de IP 200.128.79.99. Do outro lado, observa-se o servidor, que tem sua chamada representada na Figura 31, fazendo uso exclusivamente do parâmetro "-s".

Figura 30 - Chamado ao binário do IPERF

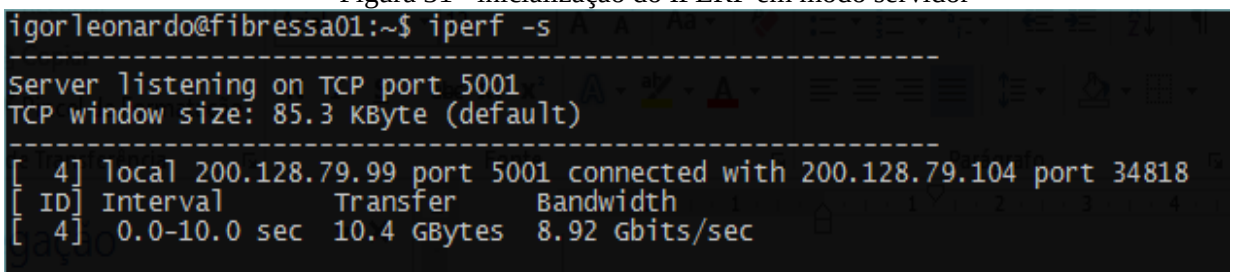




Fonte: Autoria Própria (2014).

A Figura 31 ilustra a execução do IPERF no lado do servidor. Nela pode-se observar o exato momento em que o cliente se conecta, registrando a taxa de transferência, intervalo entre os pacotes e a largura de banda.

Figura 31 - inicialização do IPERF em modo servidor



Fonte: Autoria Própria (2014).

No momento em que o IPERF cliente encerra sua execução, a base de dados testFIBREMP, de mesmo nome do identificador do experimento, se torna disponível, permitindo o dump da base completa, execução de *queries*, e outras operações permitidas pelo SQLITE. Uma observação que vale a pena ser feita, é que o OML *Server*, durante a criação da base de dados, cria as tabelas referentes aos *measurement points*, adicionado o prefixo igual ao nome da aplicação definido no arquivo ".in", seguindo de um "_", ou seja, no experimento em questão, que possui como definição de nome da aplicação o termo "IPERF", terá como tabela, para o MP "FibreMP", o nome "IPERF_fibremp" e para o MP connection, o nome "IPERF_connection".

O Quadro 7 traz o conteúdo que foi coletado pelo MP "fibremp", já armazenado na tabela "IPERF_fibremp" do OML *Server*, exibida aqui meramente com fins didáticos. Aqui é possível observar que além dos campos definidos durante a definição do MP, existem campos que não foram especificados pelo experimentador (oml_sender_id, oml_seq_oml_ts_client, oml_ts_server), mas que são adicionados automaticamente pelo OML *Server* para controle próprio.

Quadro 7 - Tabela IPERF_FibreMP, contendo os dados do experimento realizado

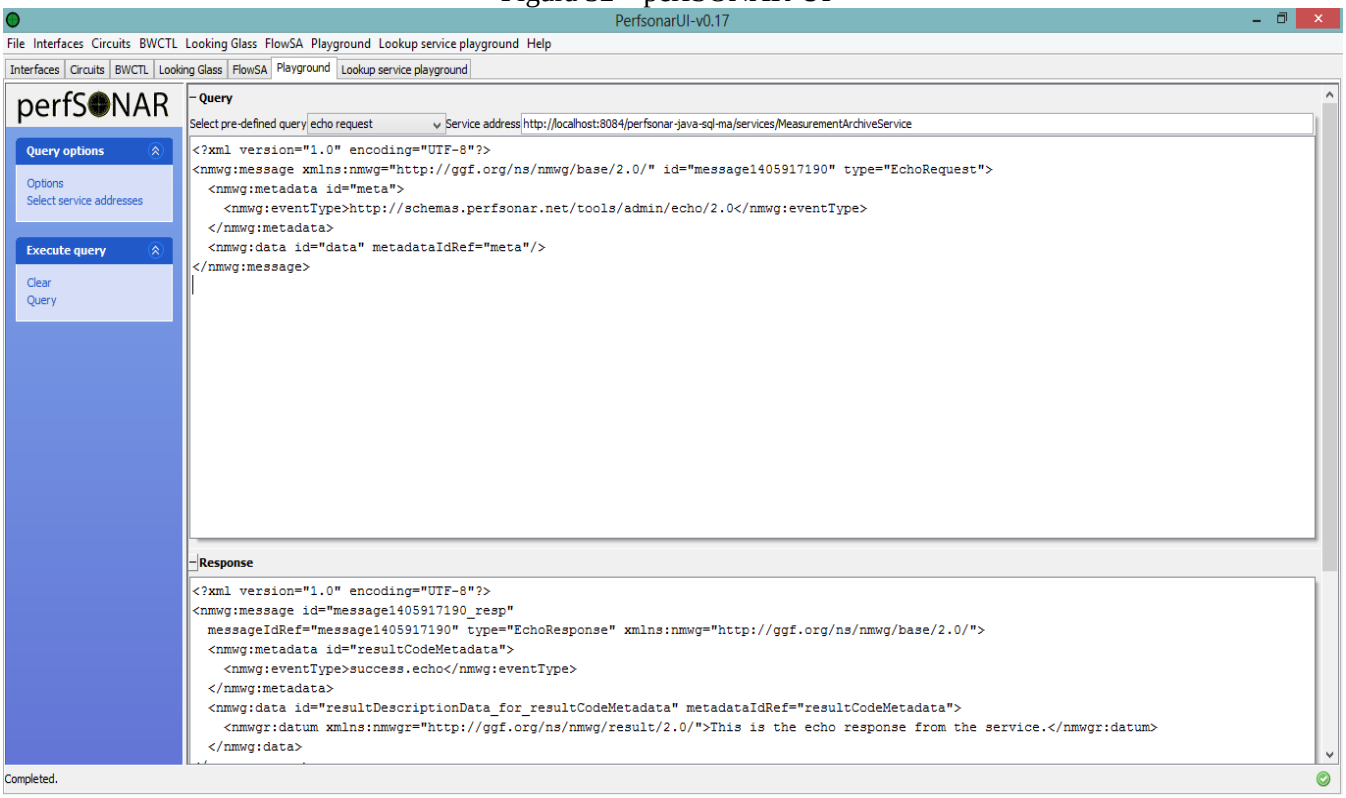
oml_sender_id	oml_seq	oml_ts_client	oml_ts_server	pid	connection_id	begin_interval	end_interval	size
1	1	104.844.899.848.104	1.048.553	19483	4	0.0	1.0	38909548
1	2	205.305.799.841.881	2.053.128	19483	4	1.0	2.0	-69103
1	3	304.641.299.694.777	3.046.482	19483	4	2.0	3.0	-68404
1	4	404.970.999.807.119	4.049.781	19483	4	3.0	4.0	-69102
1	5	504.423.199.594.021	5.044.304	19483	4	4.0	5.0	-68404
1	6	605.064.299.702.644	6.050.719	19483	4	5.0	6.0	-69102
1	7	704.511.299.729.347	7.045.186	19483	4	6.0	7.0	-68404
1	8	804.779.998.958.111	804.787	19483	4	7.0	8.0	-69102
1	9	905.013.599.991.798	9.050.193	19483	4	8.0	9.0	-69102

Fonte: Autoria Própria (2014).

5.3 O ACESSO AOS DADOS VIA OML MDIP

Com a execução do experimento finalizada, há a garantia de disponibilidade de todos os dados coletados. Dessa forma, o OML MDIP entra em ação, acessando a base especificada (via *queries* NMWG), sendo capaz de entregar todos os dados solicitados de forma que não seja preciso que o usuário tenha nenhum domínio sobre a estrutura da OML. Para tanto, é necessário fazer uso de uma aplicação compatível com o perfSONAR e nesta validação, o cliente utilizado para gerar as requisições ao OML MDIP foi o perfSONAR-UI, uma ferramenta, desenvolvida pela equipe da Internet2, para execução de testes, que permite realizar requisições aos módulos do perfSONAR, como, por exemplo, o SQLMA. Essa ferramenta, ilustrada pela Figura 32, possibilita que o usuário realize diversos tipos de testes, desde a execução de queries "pré prontas", como testes de "*echo request*", consultas BWCTL, além de outros, bem como consultas completamente customizadas pelos usuários, utilizando a seção "*Playground*" do aplicativo. Esta também foi a única seção utilizada nele.

Figura 32 – perfSONAR-UI



Fonte: Autoria Própria (2014).

A Figura 32, traz o perfSONAR-UI (UI), já na seção de *playgroung*. Na figura, observa-se, uma série de menus para acesso às diversas funções do software. Mais acima da figura, do lado esquerdo, são apresentadas duas caixas agrupadoras, uma indicando opções para a query (*Query option*) e a outra (*Execute Query*) indica funções para manipulação direta

da consulta. Na seção de query, mais importante da tela em questão, pode-se escolher uma das consultas pré-definidas ou, como foi o caso, indicar que a consulta utilizada será produzida pelo próprio utilizador do programa. Ao lado, no campo "*Service Address*", deve-se preencher com o endereço do *measurement archive* que será utilizado, neste caso, o hospedeiro no qual o OML MDIP está sendo executado. Há ainda duas grandes caixas de texto, sendo o primeiro, mais acima, para inserção da requisição/consulta e o segundo para a exibição da resposta.

Para a realização das consultas, é necessário construir um XML de requisição, seguindo os padrões do NMWG, para que esta seja submetida ao OML MDIP, por meio do perfSONAR-UI, que executa tal procedimento por meio do clique do botão "Query", à esquerda de sua interface.

A requisição utilizada no teste, é demonstrada na Figura 33. A ideia aqui, como já mencionado, é fazer uma consulta ao servidor da OML, especificando um filtro para a obtenção de resultados específicos. Na query da Figura 33, pretende-se consultar a base "testFIBREMP.sq3", trazendo somente os resultados que possuírem o campo "*size*" igual a "-69102". Dessa forma, o conteúdo da query indica que: As linhas de 1 a 5 são obrigatórias do NMWG e especificam o começo da requisição (linha 2) e o conjunto de métricas que serão utilizadas (linhas de 3 a 5).

Figura 33 - Requisição SOAP no formato NMWG

```
01:<?xml version="1.0"?>
02:<nmwg:message type="SetupDataRequest" id="setupDataRequest1"
03:               xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/"
04:               xmlns:nmwgt="http://ggf.org/ns/nmwg/topology/2.0/">
05:               xmlns:omlmdip="http://ggf.org/ns/nmwg/tools/omlmdip/"
06:  <nmwg:metadata id="testFibreMP.sq3">
07:    <omlmdip:parameters id="filters">
08:          <nmwg:parameter name="size">-69102</nmwg:parameter>
09:    </omlmdip:parameters>
10:    <nmwg:eventType>http://ggf.org/ns/nmwg/tools/omlmdip</nmwg:eventType>
11:  </nmwg:metadata>
12:  <nmwg:data id="d1" metadataIdRef="testFibreMP.sq3" />
13:</nmwg:message>
```

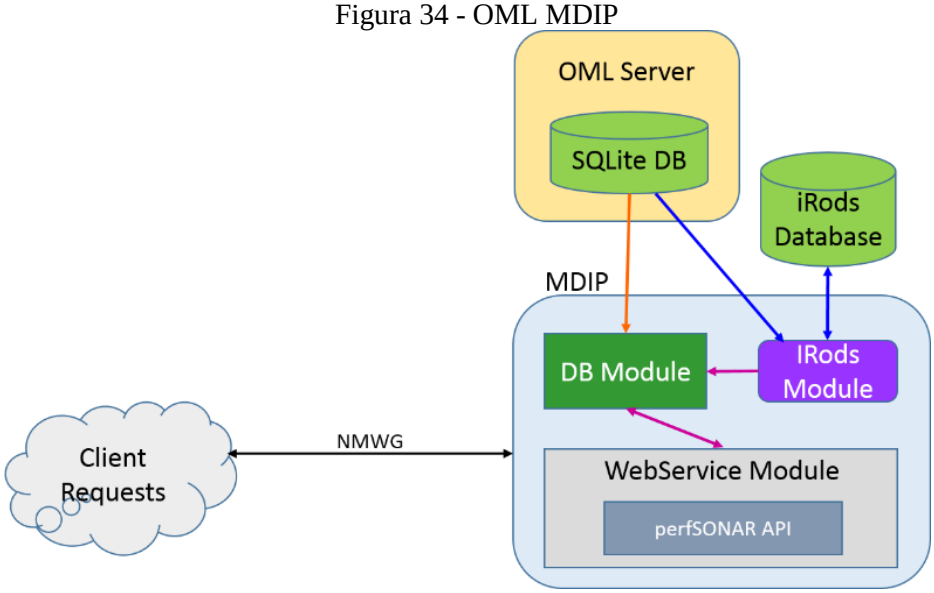
Fonte: Autoria Própria (2014).

Ainda analisando a Figura 33, as linhas de 6 a 11 indicam parâmetros específicos sobre os dados de medição de um experimento. A linha 6 representa a base de dados a ser consultada, neste caso, o "testFibreMP.sq3". Uma das estratégias do OML MDIP foi identificar o arquivo da base de dados (no SQLite, cada base de dados é equivalente a um arquivo e cada experiemento possuirá uma base exclusiva) como sendo o “id” do campo “Metadata”, obrigatório no NMWG. Na linha 7 há a definição do elemento XML que representa os filtros a serem utilizados na query de banco de dados, indicada pelo termo "*filter*". A linha 8 representa o filtro utilizado para restringir a consulta. Nesta demonstração, somente um filtro é utilizado, no entanto, mais de um poderia ser utilizado. A requisição em

questão pretende obter informações contidas na base de testes “testFibreMP.sq3”, utilizando somente o “*size*” como filtro da consulta. A linha 09 encerra os parâmetros do MDIP, enquanto que a 10 indica o tipo de evento (*EventType*) disparado, neste caso um evento do tipo "omlmdip[19]". A linha 12 define a identificação do campo "data", que representa o campo de resposta do NMWG.

Com a submissão feita através do UI, representado na Figura 34 como *Client Requests*, este abre uma conexão com o *webservice* na porta 8084, que recebe a requisição, validando e interpretando todo seu conteúdo. Com os parâmetros de consulta e nome do BD obtidos, essas informações são repassadas para o *DB Module*, que gera a query do banco e a submete à base especificada, realizando a consulta e recebendo os dados solicitados de acordo com a filtragem especificada no XML.

Figura 34 - OML MDIP



Fonte: Autoria Própria (2014).

Após a obtenção dos dados pelo *DB Module*, ele entrega os dados de reposta ao *webservice module*, para construção do XML de resposta (Resposta SOAP, no formato NMWG), sendo posteriormente enviado como resposta ao requisitante.

A resposta recebida pelo UI é representada pela Figura 35. Nela, observa-se o esperado, ou seja, obter todos os intervalos que tivessem o *size* igual a "-69102". De forma detalhada, o XML de retorno indica: das linhas de 1 a 9, observa-se o campo de Metadata idêntico ao da requisição (característica do perfSONAR e NMWG). A partir da linha 10,

[19] O EventType do NMWG define o tipo de medição que será utilizado para aquela seção. O OML MDIP utiliza o "omlmdip", criado para atender as necessidades do OML MDIP.

encontra-se a reposta à requisição realizada, tendo como retorno o resultado especificado no XML de requisição.

Figura 35 - Reposta SOAP no formato NMWG

```
01:<?xml version="1.0" encoding="UTF-8"?>
02:<nmwg:message id="setupDataRequest1_resp"
03:  messageIdRef="setupDataRequest1" type="SetupDataResponse" xmlns:nmwg="http://ggf.org/ns/nmwg/base/2.0/">
04:  <nmwg:metadata id="testFibreMP.sq3">
05:    <omlmdip:parameters id="filters" xmlns:omlmdip="http://ggf.org/ns/nmwg/tools/omlmdip/">
06:      <nmwg:parameter name="size">-69102</nmwg:parameter>
07:    </omlmdip:parameters>
08:    <nmwg:eventType>http://ggf.org/ns/nmwg/tools/omlmdip</nmwg:eventType>
09:  </nmwg:metadata>
10:  <nmwg:data id="127.0.0.1.3a8c6042:1475734863c:-7ff2">
11:    <omlmdip:datum begin_interval="3.0" end_interval="4.0" size="-69102" xmlns:omlmdip="http://ggf.org/ns/nmwg/tools/omlmdip/"/>
12:    <omlmdip:datum begin_interval="5.0" end_interval="6.0" size="-69102" xmlns:omlmdip="http://ggf.org/ns/nmwg/tools/omlmdip/"/>
13:    <omlmdip:datum begin_interval="7.0" end_interval="8.0" size="-69102" xmlns:omlmdip="http://ggf.org/ns/nmwg/tools/omlmdip/"/>
14:    <omlmdip:datum begin_interval="8.0" end_interval="9.0" size="-69102" xmlns:omlmdip="http://ggf.org/ns/nmwg/tools/omlmdip/"/>
15:  </nmwg:data>
16:</nmwg:message>
```

Fonte: Autoria Própria (2014).

6 CONCLUSÕES

A necessidade por prover uma melhor infraestrutura de rede para a internet, seja otimizando protocolos ou criando novos, é real e cada vez mais a comunidade de pesquisadores e empresas vem desenvolvendo pesquisas nessa área, de forma que tanto as empresas como os usuários possam se beneficiar com os resultados alcançados.

No sentido de prover uma infraestrutura para testes de novos protocolos e/ou modificações dos existentes, as redes para experimentação se tornaram definitivas nesse quesito. No entanto, sua rápida popularização em paralelo a sua grande demanda, acabou por estimular o surgimento de diversas tecnologias com esse mesmo fim, criando diversos ambientes incapazes de comunicarem entre si.

Com o objetivo de criar uma solução que pudesse abranger diversas tecnologias, integrando CMFs, o *testbed* FIBRE se propôs a unir o Protogeni, o OFELIA e o OMF, CMFs capazes de prover um ambiente de larga escala, flexível e robusto, gerenciando de forma completa recursos físicos e virtuais.

A integração de tecnologias distintas demandou uma metodologia de padronização, a fim de integrá-las em todo o processo de um experimento. Para atender a essa demanada, foram propostos os MDIPs, que de uma forma simplificada, é capaz de "traduzir" toda a comunicação de um CMF, para uma "linguagem" comum, integrada.

Tendo em vista a padronização e unificação dos dados de medição, surgiu o OML MDIP, uma proposta focada no uso da OML em conjunto com o perfSONAR, para possibilitar ao experimentador exportar os dados de medição, obtidos da realização do experimento, de maneira uniforme, facilitando a portabilidade desses dados, além de torná-los de fácil interpretação.

Baseado no perfSONAR SQLMA, também desenvolvido em JAVA, o desenvolvimento do OML MDIP demandou o uso de diversas tecnologias, tais como *WebService* com SOA, XML, linguagem JAVA, além de redes de computadores e infraestrutura. Sua ideia nasceu a partir do conceito dos MDIPs, tendo se concretizado efetivamente com a implementação deste.

Da forma como foi elaborado, o OML MDIP se utiliza de um esquema NMWG, criado exclusivamente para definir a base de dados dos experimentos compatíveis com o *testbed* FIBRE, descrevendo cada uma das métricas a serem utilizadas no experimento.

O OML MDIP possui a capacidade de traduzir os dados de medição capturados pela OML, para uma linguagem padronizada, aplicando métricas pré definidas ao que deve ser monitorado, como demonstrado nos capítulos 4 e 5 desta dissertação, o OML MDIP se caracateriza como uma alternativa viável para integração de tecnologias diferentes, com foco inicial na OML.

A adoção de uma metodologia rígida, porém ampla, para definir as métricas de medição, pode não ser bem aceito num primeiro momento, visto que dessa forma, a liberdade oferecida pela OML, é parcialmente coibida. Mas com a compreensão do objetivo, por parte do usuário, e seu real benefício, que é poder ter todos os seus dados de medição exportados para um ambiente homogêneo, principalmente quando da realização de experimentos em diversos CMFs, fará do OML MDIP uma ferramenta bem difundida.

6.1 TRABALHOS FUTUROS

Como sugestão para trabalhos futuros, existe a necessidade de acompanhar a evolução do perfSONAR, tornado a aplicação compatível com o JASON e webservices REST. Além disso, a fim de flexibilizar a compatibilização do experimento com o *testbed* FIBRE, é importante adotar uma meotodologia mais autônoma, no que se refere à contextualização das métricas, permitindo que o usuário/experimentador possa definir métricas que sejam de sua necessidade, mas que não estejam contempladas no escopo proposto.

REFERENCIAS

FIBRE. *Future Internet Between Brazil and Europe*. 2014. Disponível em: <http://www.fibre-ict.eu/> Acesso em: 2 jan. 2014.

FIBRE WIKI. Future Internet testbeds/experimentation between BRazil and *Europe*. 2014. Disponível em: <https://wiki.rnp.br/display/fibre> Acesso em: 7 jan. 2014.

GEANT. *perfSONAR MDM Adminstrator's Guide*. 2014. Disponível em: <https://forge.geant.net/forge/display/perfsonar/perfSONAR+MDM+Administrator%27s+Guide>. Acesso em: 25 jan. 2014.

GENI. *Exploring Networks of the Future*. Disponível em: <https://www.geni.net> Acesso em: 2 jan. 2014.

HOHLENWEGER, Thiago Souza. *Estratégia para armazenamento transparente dos dados de experimentos baseado no irods para arcabouço de controle e monitoramento da rede FIBRENET*. 2014. Dissertação (Mestrado em Sistemas e Computação)-UNIFACS, Salvador, 2014.

INTERNET2. "*What is perfSONAR?*". 2014. Disponível em: <http://www.perfsonar.net> Acesso em: 15 jan. 2014.

IPERF. "*What is IPERF*". Disponível em: <https://IPERF.fr> Acesso em: 28 jan. 2014.

IRODS WIKI. i*RODS: Data Grids, Gigital Libraries, Persistent Archives and Real Time Data Systems*. 2014. Disponível em: <https://www.iRODS.org> Acesso em: 5 fev. 2014.

MEHANI, Oliver et al.. *An Instrumentaion Framework for the Critical Task of Measurement Collection in the Future Internet*. Polônia: Computer Networks, 2014.

NICTA. *OMF – Unlock Your Experiments*. 2014. Disponível em: <http://mytestbed.net> Acesso em: 2 jan. 2014.

OFELIA FP7 Project. *The Ofelia Control Framework.* 2014. Disponível em: <http://fp7-ofelia.github.io/ocf/> Acesso em: 18 jan. 2014.

ONELAB. *Emulab – Network Emulation Testbed*. 2014. Disponível em: <https://www.emulab.net.> Acesso em: 22 jan. 2014.

PINHEIRO, Marcelo et al. *An Intrumentation and Measurement Architecture Supporting Multiple Control Monitoring Frameworks.* In: WORKSHOP ON EXPERIMENTAL RESEARCH ON THE FUTURE INTERNET (WPEIF) 3., 2012, Ouro Preto, Minas Gerais. *Anais...* 2012.

PINHEIRO, Marcelo et al. *FIBRE-BR Instrumentation and Monitoring (I&M) Tools*. In: WP2 MEETING, 2012, São Paulo. *Proceedings...* São Paulo, 2012.

ProtoGENI. *ProtoGENI Project.* 2014. Disponível em: <http://groups.geni.net/geni/wiki/> Acesso em: 14 jan. 2014.

SINGH, Manpreet et al. *ORBIT measurements framework and library (OML):* Motivations, design, implementation, and features. Itália: TridentCom., 2005.

SURUAGY, José; MARTINS, Joberto. Monitoring and Federation Progress Report & Open Issues. In: FIBRE GENERAL ASSEMBLY MEETING, 2., 2013, Barcelona, Espanha. *Proceedings...* 2013.

TWGRID. *iRODS Overview*. 2014. Disponível em: <http://www.twgrid.org/en/index.php?option=com_content&task=view&id=192&Itemid=89> Acesso em: 25 jan. 2014.

W3SCHOOLS. What is SOAP?. 2014. Disponível em: <http://www.w3schools.com> Acesso em: 12 jan. 2014.

ZURAWSKI, Jason; SWANY, Martin; GUNTER, Dan. A Scalable Framework for Representation and Exchange of Network Measurements. In: TRIDENTCOM, 2006, Barcelona, Espanha. *Proceedings...* 2006

Apêndice A.1 – Métricas do *Measurement Point* FibreMP

Atributos	Descrição
Alias	Nome alternativo ao original
DSCP	Diferentiated Services Code Point (IPV4)
OID	Object identifier
SNMPCommunity	Comunidade SNMP
SNMPVersion	Versão SNMP
Start_time	Tempo inicial
Address	Endereço IP
anonymizationLevel	Nivel de anonimização da informação
argument	Argumento genérico utilizado para informar um parâmetro à aplicação
arguments	Lista de argumentos genéricos utilizados para informar um parâmetro à aplicação
authRealm	Define um realm (domínio) para a autenticação
bufferLength	Tamanho do Buffer
bytes	Quantidade de Bytes
capacity	Capacidade em bytes
characteristics	Caracteristicas do objeto
classOfService	Classe do serviço
command	Comando informado via cmd
count	Contador genérico
description	Descrição do objeto
duplicates	Flag indicativo de duplicação, para pacotes ping
duration	Tempo de duração de um determinado objeto
end_time	Tempo de encerramento do processo
event_number	Número identificador do evento
event_specification	Especificação do domínio
event_type	Tipo de evento
exponential	Métrica utilizada pelo OWD
firstTtl	Métrica utilizada pelo traceroute
fixed	Métrica utilizada pelo OWD
flow_h_filter	Ttcpdump (PCAP) compatible filter expression.
flow_p_filter	Packet payload filtering regular expression
flow_s_thre	Specify sampling parameters, where flow_s_type may be deterministic (or periodic) or probabilistic
flow_s_type	Specify sampling parameters, where flow_s_type may be deterministic (or periodic) or probabilistic
granularity	Nível de granularidade do flow
hop	Identioficador do nó
hospedeironame	Nome do hospedeiro
ifAddress	Endereço IP da interface de rede
ifDescription	Descrição da interface de rede
ifHospedeironame	Identificador do hospedeiro baseado na interface de rede
ifIndex	Índice da interface de rede
ifName	Nome da interface de rede
interval	Intervalo de tempo (unidade definida pelo experimentador)
begin_interval	Instante de tempo de tempo inicial (unidade definida pelo experimentador)
end_interval	Instante de tempo de tempo final (unidade definida pelo experimentador)
interval_sec	Intervalo de tempo em segundos
interval_usec	Intervalod e tempo em milissegundos
jitter	Tepo de atraso do pacote
latest	Identificador do ultimo pacote recebido
linkIdRef	Identificador de referência de link
loss	Indicador de perda de pacotes

lossPercent	Percentagem de perda
loss_rate	Taxa de perda de pacotes
mask	Máscara e rede
maxRtt	Round Trip Time máximo
maxTtl	TTL máximo
measurementOffset	Offsert de medição
measurementPeriod	Período de medição
medianRtt	RTT médio
messageIdRef	Identificador de referência da mensagem
metadataidRef	Identificador de referência da metadado
minRtt	RTT mínimo
netflowversion	Versão utilizada do netflow
networkIdRef	Identificador de referência da rede
numBytes	Número de bytes do pacote
numBytesUnits	Número de unidades de bytes do pacote
numberOfTests	Númeroq uantificador de testes
outOforder	Flag indicativo de falta de ordenação
packetInterval	Intervalod e tempo do pacote
packetSize	Tamanho do pacote
packetTTL	TTL do pacote
pathIdRef	Identificador d referência do caminho de rede
pause	Flasg de pausa da aplicação
pkts	Número de pacotes
port	Identificador de porta
protocol	Protocolo utilizado
protocols	Protocolos utilizados
queryNum	Número identificador da query
scheduleInterval	Intervalo de agendamento
sent	Flag indicativo de envio
seqNum	Número de sequencia do pacote
start_time	Tempo de início do processo
subnet	Subnet utilizada
supportedEventType	Tipo de EventType suportado
syncronized	Flasg indicador de sincronização
timeDuration	Tempo de duração do processo
timeType	Tipo de tempo utilizado
timeValue	Valor de tempo utilizado
timeout	Flasg indicativo de timeout
timestamp	Timestamp
transportProtocol	Protocolo de transporte
ttl	Time to live
type	Variável indicativo de tipo (genérica)
value	Variável indicativa de valor (genérica)
valueUnits	Variável indicativa de unidades de valor (genérica)
waitTime	Tempo de espera
windowSize	Tamanho da janela utilizada
windowSizeRequest	Tamanho da janela requisiada

Apêndice A.2 – Esquema NMWG da OMLMdip

```
# ###########################################################
# Namespace definitions
# ###########################################################
namespace nmwg = "http://ggf.org/ns/nmwg/base/2.0/"
namespace nmwgr = "http://ggf.org/ns/nmwg/result/2.0/"
namespace omlmdip = "http://ggf.org/ns/nmwg/tools/omlmdip/"


# ###########################################################
# Include additional functionality from other files
# ###########################################################
include "nmtopo.rnc"
include "nmtopo_ver3.rnc"
include "result.rnc"
include "nmbase.rnc" {
      Metadata |= OmlMdipMetadata
      Data |= OmlMdipData
}


OmlMdipMetadata =
      element nmwg:metadata {
            Identifier &
            MetadataIdentifierRef? &
            OmlMdipMetadataContent
      }

OmlMdipMetadataBlock =
      OmlMdipSubject? &
      (
            OmlMdipParameters |
            Parameters
      )?

OmlMdipMetadataContent =
      (
            OmlMdipMetadataBlock |
            FilterMetadataBlock
      ) &
      EventType* &
      Key?


OmlMdipSubject =
      element omlmdip:subject {
            Identifier &
            MetadataIdentifierRef? &
            (
                  EndpointPair |
                  L4EndpointPair
            )
      }




OmlMdipParameters =
      element omlmdip:parameters {
            Identifier &
            OmlMdipParameter+
      }

OmlMdipParameter =
      element nmwg:parameter {
            attribute name { "Alias" | "DSCP" | "OID" | "SNMPCommunity" | "SNMPVersion" | "Start_time" |
                        "Address" | "anonymizationLevel" | "argument" | "arguments" | "authRealm" |
                        "bufferLength" | "bytes" | "capacity" | "characteristics" | "classOfService" |
                        "command" | "count" | "description" | "duplicates" | "duration" | "end_time" |
```

```
                    "event_number" | "event_specification" | "event_type" | "exponential" |
                    "firstTtl" | "fixed" | "flow_h_filter" | "flow_p_filter" | "flow_s_thre" |
                    "flow_s_type" | "granularity" | "hop" | "hospedeironame" | "ifAddress" |
                    "ifDescription" | "ifHospedeironame" | "ifIndex" | "ifName" | "interval" |
                    "begin_interval" | "end_interval" | "interval_sec" | "interval_usec" |
                    "jitter" | "latest" | "linkIdRef" | "loss" | "lossPercent" | "loss_rate" |
                    "mask" | "maxRtt" | "maxTtl" | "measurementOffset" | "measurementPeriod" |
                    "medianRtt" | "messageIdRef" | "metadataidRef" | "minRtt" | "netflowversion" |
                    "networkIdRef" | "numBytes" | "numBytesUnits" | "numberOfTests" | "outOforder" |
                    "packetInterval" | "packetSize" | "packetTTL" | "pathIdRef" | "pause" | "pkts" |
                    "port" | "protocol" | "protocols" | "queryNum " | "scheduleInterval" | "sent" |
                    "seqNum " | "start_time" | "subnet" | "supportedEventType" | "syncronized" |
                    "timeDuration" | "timeType" | "timeValue" | "timeout" | "timestamp" |
                    "transportProtocol" | "ttl" | "type" | "value" | "valueUnits" | "waitTime" |
                    "windowSize " | "windowSizeRequest" } &
        (
            attribute value { text } |
            text
        )
    }



OmlMdipData =
    element nmwg:data {
        Identifier &
        MetadataIdentifierRef? &
        (
            (
                Metadata* |
                OmlMdipMetadata*
            ) |
            (
                OmlMdipCommonTime+ &
                (
                    OmlMdipDatum* |
                    ResultDatum*
                )
            ) |
            (
                OmlMdipDatum* |
                ResultDatum*
            ) |
            Key*
        )
    }



OmlMdipCommonTime =
    element nmwg:commonTime {
        Type &
        (
            TimeStamp |
            (
                StartTime &
                (
                    EndTime |
                    Duration
                )
            )
        ) &
        (
            OmlMdipDatum* |
            ResultDatum*
        )
    }



OmlMdipDatum =
```

```
element omlmdip:datum {
      attribute   Alias       { xsd:string }? &
      attribute   DSCP        { xsd:string }? &
      attribute   OID         { xsd:string }? &
      attribute   SNMPCommunity   { xsd:string }? &
      attribute   SNMPVersion     { xsd:string }? &
      attribute   Start_time      { xsd:string }? &
      attribute   Address   { xsd:string }? &
      attribute   anonymizationLevel { xsd:string }? &
      attribute   argument { xsd:string }? &
      attribute   arguments       { xsd:string }? &
      attribute   authRealm       { xsd:string }? &
      attribute   bufferLength    { xsd:string }? &
      attribute   bytes       { xsd:string }? &
      attribute   capacity    { xsd:string }? &
      attribute   characteristics     { xsd:string }? &
      attribute   classOfService      { xsd:string }? &
      attribute   command { xsd:string }? &
      attribute   count       { xsd:string }? &
      attribute   description     { xsd:string }? &
      attribute   duplicates{ xsd:string }? &
      attribute   duration    { xsd:string }? &
      attribute   end_time    { xsd:string }? &
      attribute   event_number        { xsd:string }? &
      attribute   event_specification   { xsd:string }? &
      attribute   event_type      { xsd:string }? &
      attribute   exponential     { xsd:string }? &
      attribute   firstTtl    { xsd:string }? &
      attribute   fixed       { xsd:string }? &
      attribute   flow_h_filter       { xsd:string }? &
      attribute   flow_p_filter       { xsd:string }? &
      attribute   flow_s_thre         { xsd:string }? &
      attribute   flow_s_type         { xsd:string }? &
      attribute   granularity         { xsd:string }? &
      attribute   hop         { xsd:string }? &
      attribute   hospedeironame      { xsd:string }? &
      attribute   ifAddress { xsd:string }? &
      attribute   ifDescription       { xsd:string }? &
      attribute   ifHospedeironame    { xsd:string }? &
      attribute   ifIndex     { xsd:string }? &
      attribute   ifName      { xsd:string }? &
      attribute   interval    { xsd:string }? &
      attribute   begin_interval      { xsd:string }? &
      attribute   end_interval        { xsd:string }? &
      attribute   interval_sec        { xsd:string }? &
      attribute   interval_usec       { xsd:string }? &
      attribute   jitter      { xsd:string }? &
      attribute   latest      { xsd:string }? &
      attribute   linkIdRef { xsd:string }? &
      attribute   loss        { xsd:string }? &
      attribute   lossPercent         { xsd:string }? &
      attribute   loss_rate   { xsd:string }? &
      attribute   mask        { xsd:string }? &
      attribute   maxRtt      { xsd:string }? &
      attribute   maxTtl      { xsd:string }? &
      attribute   measurementOffset  { xsd:string }? &
      attribute   measurementPeriod { xsd:string }? &
      attribute   medianRtt       { xsd:string }? &
      attribute   messageIdRef        { xsd:string }? &
      attribute   metadataidRef       { xsd:string }? &
      attribute   minRtt      { xsd:string }? &
      attribute   netflowversion      { xsd:string }? &
      attribute   networkIdRef        { xsd:string }? &
      attribute   numBytes            { xsd:string }? &
      attribute   numBytesUnits       { xsd:string }? &
      attribute   numberOfTests       { xsd:string }? &
      attribute   outOforder          { xsd:string }? &
      attribute   packetInterval      { xsd:string }? &
      attribute   packetSize          { xsd:string }? &
      attribute   packetTTL           { xsd:string }? &
      attribute   pathIdRef{ xsd:string }? &
      attribute   pause       { xsd:string }? &
      attribute   pkts        { xsd:string }? &
      attribute   port        { xsd:string }? &
      attribute   protocol    { xsd:string }? &
      attribute   protocols   { xsd:string }? &
```

```
    attribute   queryNum              { xsd:string }? &
    attribute   scheduleInterval      { xsd:string }? &
    attribute   sent          { xsd:string }? &
    attribute   seqNum    { xsd:string }? &
    attribute   start_time{ xsd:string }? &
    attribute   subnet      { xsd:string }? &
    attribute   supportedEventType              { xsd:string }? &
    attribute   syncronized           { xsd:string }? &
    attribute   timeDuration          { xsd:string }? &
    attribute   timeType { xsd:string }? &
    attribute   timeValue{ xsd:string }? &
    attribute   timeout     { xsd:string }? &
    attribute   timestamp             { xsd:string }? &
    attribute   transportProtocol     { xsd:string }? &
    attribute   ttl           { xsd:string }? &
    attribute   type          { xsd:string }? &
    attribute   value         { xsd:string }? &
    attribute   valueUnits            { xsd:string }? &
    attribute   waitTime { xsd:string }? &
    attribute   windowSize            { xsd:string }? &
    attribute   windowSizeRequest { xsd:string }? &?
}
```

Apêndice B.1 – Classe do DB Module - SQLTypeMAServiceEngine

```
package org.perfsonar.service.measurementArchive.SQLType;

import java.util.Collection;
import java.util.Iterator;
import java.util.Set;


import org.ggf.ns.nmwg.base.v2_0.Data;
import org.ggf.ns.nmwg.base.v2_0.Element;
import org.ggf.ns.nmwg.base.v2_0.EventType;
import org.ggf.ns.nmwg.base.v2_0.Key;
import org.ggf.ns.nmwg.base.v2_0.Message;
import org.ggf.ns.nmwg.base.v2_0.Metadata;
import org.ggf.ns.nmwg.base.v2_0.Parameter;
import org.ggf.ns.nmwg.base.v2_0.Parameters;
import org.ggf.ns.nmwg.base.v2_0.Subject;

import org.perfsonar.base.auxiliary.ComponentNames;
import org.perfsonar.base.auxiliary.components.configuration.ConfigurationComponent;
import org.perfsonar.base.auxiliary.components.logger.LoggerComponent;
import org.perfsonar.base.exceptions.PerfSONARException;
import org.perfsonar.base.auxiliary.AuxiliaryComponentManager;

import org.perfsonar.service.base.engine.ActionType;
import org.perfsonar.service.base.engine.ServiceEngine;
import org.perfsonar.service.base.messages.BaseChaining;
import org.perfsonar.service.base.storage.NMWGStorageManager;
import org.perfsonar.service.base.util.ResultCodesUtil;
import org.perfsonar.service.measurementArchive.metadataConfig.MetadataConfigurationStorageManager_nmwg;


/**
 * Class implementing the Measurement Archive Service Engine which deals with
 * SQL database.
 *
 * @author romradz
 * @see org.perfsonar.service.base.engine
 */
public class SQLTypeMAServiceEngine implements ServiceEngine
{
    /**
     * Type of service engine.
     */
    private final String serviceEngineType;


    /**
     * The object to log control/debugging messages.
     */
    private final LoggerComponent logger;


    /**
     * This object provides access to the service configuration
     * (service.properties).
     */
    private final ConfigurationComponent configuration;


    /**
     * This object provides access to metadata configuration.
     */
    private final NMWGStorageManager metadataConfig;

    private JdbcStorageManager storageManager;

    private final String dbStatus;
    private final String dbStore;
```

```
/**
 * This variable is used to detect if there is more than one
 * data triggers in the request.
 */
protected boolean firstRunDone = false;


public static final String SELECT_NAMESPACE = "http://ggf.org/ns/nmwg/ops/select/2.0";

public static final String ECHO_SELF_TEST_NAMESPACE =
    "http://schemas.perfsonar.net/tools/admin/selftest/";
public static final String ECHO_XMLDB_ACCESS_TEST_NAMESPACE =
    "http://schemas.perfsonar.net/tools/admin/selftest/1.0/xmldb-access-test";
public static final String ECHO_XMLDB_CONTENT_TEST_NAMESPACE =
    "http://schemas.perfsonar.net/tools/admin/selftest/1.0/xmldb-content-test";
public static final String ECHO_RANDOM_FETCH_TEST_NAMESPACE =
    "http://schemas.perfsonar.net/tools/admin/selftest/1.0/random-fetch-test";


// ---------------------------------- constructors


public SQLTypeMAServiceEngine() throws PerfSONARException
{
    serviceEngineType = "service.MeasurementArchive.SQLType";
    logger = getLogger();
    configuration = getConfiguration();
    storageManager = new JdbcStorageManager();

    // checking if using xml db is allowed
    dbStatus = configuration.getProperty("component.ma.xmldb.db_status");
    dbStore = configuration.getProperty("service.ma.xmldb.db_store");

    metadataConfig = new MetadataConfigurationStorageManager_nmwg();
    metadataConfig.initStorage(null);
}


// ---------------------------------- methods

protected Message getData(Message request) throws PerfSONARException
{
    Message response = new Message();

    //Message keyMessage = getKey(request);

    boolean mustCopyMetadata = false;

    Collection<Metadata> collection = request.getMetadataMap().values();

    //It will just have one metadata. its id will identifie database name
    for (Metadata md : collection) {
        storageManager = new JdbcStorageManager(md);
        Message query = new Message();
        Message dMsg = storageManager.fetch(md);

        // Add data elements to response
        for (Data d : (Collection<Data>) dMsg.getDataMap().values()) {
            response.addChild(d);
            d.setMetadataIdRef(md.getMetadataIdRef());
        }
    }

    for (Data keyData : (Collection<Data>) request.getDataMap().values()) {
        // Fetch data from data store
        Metadata queryMetadata = new Metadata();
        queryMetadata.setKey(keyData.getKey());
        queryMetadata.setEventType(request.getMetadata(keyData.getMetadataIdRef()).getEventType());
        Message query = new Message();
        query.setMetadata(queryMetadata);
        //Message dMsg = storageManager.fetch(query);

        // Add metadata elements to response
        String metadataIdRef;
        if (keyData.getMetadataIdRef().isEmpty()) {
```

```
            response.addChild(queryMetadata);
            metadataIdRef = queryMetadata.getId();
        } else {
            mustCopyMetadata = true;
            metadataIdRef = keyData.getMetadataIdRef();
        }
    }

    if (mustCopyMetadata) {
        for (Metadata m : (Collection<Metadata>) request.getMetadataMap().values()) {
            response.addChild(m);
        }
    }

    return response;
}


protected Message storeData(Message request) throws PerfSONARException
{
    Message response = new Message();

    for (Data data : (Collection<Data>) request.getDataMap().values()) {
        Metadata meta = request.getMetadata(data.getMetadataIdRef());
        if (meta != null) {
            Metadata md = new Metadata();
            md.setKey(getStoreKey(meta));

            Message msg = new Message();
            msg.addChild(md);
            msg.addChild(data);

            // FIXME: This means only the last response is passed on to the client.
            // Should these responses be merged?
            response = storageManager.store(msg);
        }
    }

    return response;
}


public Message storeKey(Message request)
    throws PerfSONARException
{
    metadataConfig.store(request);
    return generateResultCodeResponse("success.ma.write", "Key has been stored");
}


protected Key getStoreKey(Metadata metadata)
    throws PerfSONARException
{
    EventType eventTypeObj = metadata.getEventType();
    if (eventTypeObj == null) {
        throw new PerfSONARException("error.ma.query",
            "SQLTypeMAServiceEngine.getStoreKey: "
                + "No eventType in the request metadata");
    }

    String eventType = eventTypeObj.getEventType().trim();
    if (eventType.isEmpty()) {
        throw new PerfSONARException("error.ma.query",
            "SQLTypeMAServiceEngine.getStoreKey: "
                + "eventType in the request metadata is empty");
    }

    Key key = metadata.getKey();

    if (key == null) {
        // no key in the request
        Message query = new Message();
        query.addChild(metadata);
        Message resp;

        try {
```

```
            resp = metadataConfig.fetch(query);
        } catch (PerfSONARException sex) {
            if (!sex.getResultCode().equals("warning.ma.metadata_configuration")) {
                throw sex;
            }
            resp = new Message();
        }

        if (!resp.getDataMap().isEmpty()) {
            // metadata already exists in xmldb

            Data data = (resp.getDataArray())[0];
            key = data.getKey();

            if (key.getParameterByName("eventType") == null) {
                key.addParameter("eventType", eventType);
            }
        } else {
            // metadata does not exist in xmldb

            Metadata metadata2 = new Metadata();
            metadata2.setSubject(metadata.getSubject());
            metadata2.setEventType(eventTypeObj);
            Parameters params = metadata.getParameters();
            if (params != null) {
                metadata2.setParameters(params);
            }

            /* Parameters are added to the key such that the data store can
             * access them. Eg. OwampDao needs to know the output format.
             */
            key = new Key();
            for (Object parameter: params.getParameterMap().values()) {
                key.addParameter((Parameter) parameter);
            }
            key.addParameter("metadataId", metadata2.getId());
            key.addParameter("eventType", eventType);

            Data data = new Data();
            data.addChild(key);
            data.setMetadataIdRef(metadata2.getId());

            Message configMessage = new Message();
            configMessage.addChild(metadata2);
            configMessage.addChild(data);

            metadataConfig.store(configMessage);
        }
    }
    return key;
}

protected Message replaceMetadataConfiguration(Message request)
    throws PerfSONARException
{
    if (!isFirstRunDone()) {
        metadataConfig.removeAll();
    }

    ((MetadataConfigurationStorageManager_nmwg) metadataConfig).storeMetadataElements(request);
    ((MetadataConfigurationStorageManager_nmwg) metadataConfig).storeDataElements(request);

    storeData(request);

    Data dataTrigger = (Data) request.getDataIterator().next();
    String dataTriggerId = "";
    if (dataTrigger != null) {
        dataTriggerId = dataTrigger.getId();
    }

    String desc = "Metadata linked with request data trigger id='" + dataTriggerId + "' has been stored.";
    return generateResultCodeResponse("success.metadata_config_replace", desc);

}
```

```
//////////////////////////////////////////////////////////////
// Methods below are inlined from old RRD MA
//


/**
 * Method which provides access the type of ServiceEngine.
 * It implements the method definition in ServiceEngine Interface
 * <p/>
 * It does not return null
 *
 * @return String containing ServiceEngine Type
 * @see org.perfsonar.service.base.engine.ServiceEngine#getType()
 */
@Override
public String getType()
{
   return this.serviceEngineType;
}


protected boolean isFirstRunDone()
{
   return firstRunDone;
}


/**
 * Method to accept requests for action. It implements
 * the method definition in ServiceEngine Interface.
 * It does not return null.
 *
 * @param actionType string specifying the type of action required
 *                   on the given input
 * @param request    object containing the request
 * @return Message object containing the response
 * @throws PerfSONARException
 * @see org.perfsonar.service.base.engine.ServiceEngine
 */
@Override
public Message takeAction(String actionType, Message request)
   throws PerfSONARException
{

   logger.debug("SQLTypeMAServiceEngine.takeAction: "
      + "Received a request of type - "
      + actionType);

   Message response;

   if (actionType.equals(ActionType.GET_KEY)) {

      // handle GET_KEY
      // message type: MetadataKeyRequest
      response = getKey(request);

   } else if (actionType.equals(ActionType.GET_DATA)) {

      // handle GET_DATA
      // message type: SetupDataRequest
      response = getData(request);

   } else if (actionType.equals(ActionType.STORE_DATA)) {

      // handle STORE_DATA
      // message type: MeasurementArchiveStoreRequest
      checkXmlDbAccessProprties();
      response = storeData(request);

   } else if (actionType.equals(ActionType.STORE_KEY)) {

      // handle STORE_KEY
      checkXmlDbAccessProprties();
      response = storeKey(request);
```

```
    } else if (actionType.equals(ActionType.STORE_METADATA_CONFIGURATION)) {

        // handle STORE_METADATA_CONFIGURATION
        // replace complete metadata configuration file
        // message type: CompleteMetadataStoreRequest
        checkXmlDbAccessProprties();
        response = replaceMetadataConfiguration(request);

    } else if (actionType.equals(ActionType.ECHO)) {

        response = getEcho(request);

    } else {

        // invalid action type

        logger.error(
            "SQLTypeMAServiceEngine.takeAction: ActionType specified: "
                + actionType + " is not supported");

        throw new PerfSONARException(
            "error.ma.action",
            "SQLTypeMAServiceEngine: "
                + "ActionType specified: "
                + actionType
                + " is not supported");

    }

    firstRunDone = true;
    return response;

}

protected void checkXmlDbAccessProprties() throws PerfSONARException
{

    if (!dbStatus.trim().equalsIgnoreCase("on")) {

        logger.info(
            "SQLTypeMAServiceEngine.checkXmlDbAccessProprties: "
                + "xmldb is not setup in the configuration "
                + "(see 'component.ma.xmldb.db_status' property)");
        throw new PerfSONARException(
            "error.ma.SQL.writing",
            "SQLTypeMAServiceEngine.checkXmlDbAccessProprties: "
                + "Reading data from xmldb is switched off in the MA "
                + "(check the configuration)");

    } else if (!dbStore.trim().equalsIgnoreCase("on")) {

        logger.info(
            "SQLTypeMAServiceEngine.checkXmlDbAccessProprties: "
                + "writing data into MA is switched off "
                + "(see 'service.ma.xmldb.db_store' property)");
        throw new PerfSONARException(
            "error.ma.SQL.writing",
            "SQLTypeMAServiceEngine.checkXmlDbAccessProprties: "
                + "Writing data or xmldb is switched off in the MA.");

    }

}


private static long seq = 0;

private static synchronized String getSeq()
{
    return Long.toString(++seq);
}

protected LoggerComponent getLogger() throws PerfSONARException
{
    try {
        return (LoggerComponent) AuxiliaryComponentManager.getInstance().
```

```
            getComponent(ComponentNames.LOGGER);
        } catch (PerfSONARException e) {
            throw new PerfSONARException(
                "error.common.no_logger",
                "SQLTypeMAServiceEngine.getLogger: "
                    + "Cannot get logger component", e);
        }
    }

    protected ConfigurationComponent getConfiguration() throws PerfSONARException
    {
        try {
            return (ConfigurationComponent) AuxiliaryComponentManager.getInstance().
                getComponent(ComponentNames.CONFIG);
        } catch (PerfSONARException e) {
            throw new PerfSONARException(
                "error.common.no_configuration",
                "SQLTypeMAServiceEngine.getConfiguration: "
                    + "Cannot get config component", e);
        }
    }

    /**
     * Returns a key for a request message.
     *
     * @param request object containing the request
     * @return Message object containing the response with a key
     * @throws PerfSONARException
     */
    protected Message getKey(Message request) throws PerfSONARException
    {
        request = mergeMetaData(request);

        Message response = new Message();

        Collection<Data> dataTriggers = request.getDataMap().values();
        for (Data data : dataTriggers) {
            try {
                Metadata metadata = request.getMetadata(data.getMetadataIdRef());
                getKeyImpl(request, metadata, response);
            } catch (NullPointerException npex) {
                throw new PerfSONARException(
                    "error.ma.query",
                    "SQLTypeMAServiceEngine.getKey: "
                        + "getting the key is impossible "
                        + "because of some problem with the format of the request; "
                        + npex.toString());
            }
        }
        return response;
    }

    /**
     * Removes base chaining (chains of metadata blocks linked by
     * metadataIdRef attribute) from the message.
     *
     * @param message object to be processed
     * @return Message processed message object
     * @throws PerfSONARException
     */
    protected Message mergeMetaData(Message message)
        throws PerfSONARException
    {
        String mergerNames;
        try {
            mergerNames = configuration.getProperty("service.ma.base_chaining_mergers");
        } catch (PerfSONARException ex) {
            return message;
        }

        if (!mergerNames.trim().isEmpty()) {
            Set<BaseChaining> baseChainingMergers =
                BaseChainingMergerFactory.getBaseChainingMergers(mergerNames);

            if (baseChainingMergers != null) {
                for (BaseChaining baseChaining : baseChainingMergers) {
```

```
                message = baseChaining.processChaining(message);
            }
        }
    }

    return message;
}

/**
 * Method used in getKey method. Helps to find a key for a request message.
 */
protected void getKeyImpl(Message request, Metadata metadata, Message response)
    throws PerfSONARException
{
    Subject subject = metadata.getSubject();
    Key key = metadata.getKey();
    EventType eventType = metadata.getEventType();

    if (subject != null) {
        String metadataIdRef = subject.getMetadataIdRef();
        if (!metadataIdRef.isEmpty()) {
            getKeyByOperationChaining(request, metadata, response);
        } else {
            getKeyFromConfigurationStore(metadata, response);
        }
    } else if (key != null || eventType != null) {
        if (key != null) {
            getKeyByIdentity(key, response);
        }
        if (eventType != null) {
            getKeyFromConfigurationStore(metadata, response);
        }
    } else {
        throw new PerfSONARException(
            "error.ma.query",
            "SQLTypeMAServiceEngine.getKeyImpl: "
                + "no subject, eventType or key element in metadata "
                + metadata.getId());
    }
}

/** Passes the query on to metadataConfig for lookup in Exist. */
private void getKeyFromConfigurationStore(Metadata metadata, Message response) throws PerfSONARException
{
    Message query = new Message();
    query.addChild(metadata);

    Message msg = metadataConfig.fetch(query);

    if (metadata.getEventType() != null) {
        String eventType = metadata.getEventType().getEventType();
        for (Object d : msg.getDataMap().values()) {
            Key key = ((Data) d).getKey();
            if (key.getParameters().getParameterMap().get("eventType") == null) {
                key.addParameter("eventType", eventType);
            }
        }
    }

    for (Object m : msg.getMetadataMap().values()) {
        response.addChild((Metadata) m);
    }

    for (Object d : msg.getDataMap().values()) {
        response.addChild((Data) d);
    }
}

/** Copies the metadata key verbatim to the result. */
private void getKeyByIdentity(Key key, Message response)
{
    Metadata metadata = new Metadata();
    metadata.setKey(key);
    metadata.setId(getSeq());
    response.addChild(metadata);
```

```
    Data data = new Data();
    data.setMetadataIdRef(metadata.getId());
    data.addChild(key);

    response.addChild(data);
}

/** Operation chaining. Only the select operation is supported. */
private void getKeyByOperationChaining(Message request, Metadata metadata, Message response)
    throws PerfSONARException
{
    if (metadata.getEventType() == null) {
        throw new PerfSONARException(
            "error.ma.query",
            "SQLTypeMAServiceEngine.getKeyImpl: "
                + "Operation chaining requires event type in metadata " + metadata.getId());
    }
    String eventType = metadata.getEventType().getEventType().trim();
    if (!eventType.equals(SELECT_NAMESPACE)) {
        throw new PerfSONARException(
            "error.ma.query",
            "SQLTypeMAServiceEngine.getKeyImpl: "
                + "eventType " + eventType + " in metadata " + metadata.getId()
                + " is not supported");
    }

    Metadata chained = request.getMetadata(metadata.getSubject().getMetadataIdRef());
    getKeyImpl(request, chained, response);

    /* Select parameters are added verbatim to the output key. When subsequently fetching data,
     * the select parameters will be interpreted and used to restrict the data set being returned.
     * @see DaoJdbcTemplate
     */
    Parameters selectParameters = metadata.getParameters();
    for (Object d : response.getDataMap().values()) {
        Key key = ((Data) d).getKey();
        if (key != null) {
            key.setParameters(selectParameters);
        }
    }
}


protected Message getEcho(Message request) throws PerfSONARException
{

    // this method will be used for advanced types of echo request

    Data data = request.getDataArray()[0];
    Metadata metadata = request.getMetadata(data.getMetadataIdRef());

    EventType eventType = metadata.getEventType();
    String eventTypeStr = eventType.getEventType().trim();

    String testName;
    try {
        testName = eventTypeStr.substring(eventTypeStr.lastIndexOf("/") + 1, eventTypeStr.length());
    } catch (Exception ex) {
        testName = "test";
    }

    Message response;

    if (eventTypeStr.equals(ECHO_XMLDB_ACCESS_TEST_NAMESPACE)
        || eventTypeStr.equals(ECHO_XMLDB_CONTENT_TEST_NAMESPACE)) {
        response = executeXmlDBSimpleTest(request, testName);
    } else if (eventTypeStr.equals(ECHO_RANDOM_FETCH_TEST_NAMESPACE)) {
        response = executeRandomFetchTest(request, testName);
    } else if (eventTypeStr.equals(ECHO_SELF_TEST_NAMESPACE + "1.0")) {
        response = executeSelfTest(request);
    } else {
        return generateResultCodeResponse(
            "error.request", "This eventType in echo request is not supported");
    }

    return response;
```

```
}


protected Message executeXmlDBSimpleTest(Message request, String testName) throws PerfSONARException
{

    String serviceType = configuration.getProperty("service.r.service_type");
    String serviceName = null;
    try {
        serviceName = configuration.getProperty("service.name");
    } catch (PerfSONARException ignored) {
    }
    if (serviceName == null) {
        serviceName = "pS" + serviceType;
    }

    Message response;
    Data data;
    String eventTypeStr;

    try {

        response = metadataConfig.fetch(request);
        eventTypeStr = ECHO_SELF_TEST_NAMESPACE + serviceType + "/" + serviceName + "/" + testName + "/success/1.0";
        data = response.getDataArray()[0];

    } catch (PerfSONARException psEx) {

        eventTypeStr = ECHO_SELF_TEST_NAMESPACE + serviceType + "/" + serviceName + "/" + testName + "/failure/1.0";
        return generateResultCodeResponse(eventTypeStr, psEx.getResultDescription());
    }

    EventType eventType = new EventType();
    eventType.setEventType(eventTypeStr);

    Metadata metadata = new Metadata();
    metadata.setEventType(eventType);

    response.setMetadata(metadata);
    data.setMetadataIdRef(metadata.getId());

    return response;
}


protected static final String CODE_RESPONSE_1 = "Fetching data from the relational database is correct.";

protected Message executeRandomFetchTest(Message request, String testName) throws PerfSONARException
{

    String serviceType = configuration.getProperty("service.r.service_type");
    String serviceName = null;
    try {
        serviceName = configuration.getProperty("service.name");
    } catch (PerfSONARException ignored) {
    }
    if (serviceName == null) {
        serviceName = "pS" + serviceType;
    }

    Message keyMessage;
    try {
        keyMessage = metadataConfig.fetch(request);
    } catch (PerfSONARException psEx) {
        return generateResultCodeResponse(
            ECHO_SELF_TEST_NAMESPACE + serviceType + "/" + serviceName + "/" + testName + "/failure/1.0",
            psEx.getResultDescription());
    }

    Data keyData;
    try {
        keyData = keyMessage.getDataArray()[0];
    } catch (Exception ex) {
        return generateResultCodeResponse(
            ECHO_SELF_TEST_NAMESPACE + serviceType + "/" + serviceName + "/" + testName + "/failure/1.0",
            "No data elements in the metadata configuration");
```

```
    }
    Key key = keyData.getKey();
    if (key == null) {
       return generateResultCodeResponse(
          ECHO_SELF_TEST_NAMESPACE + serviceType + "/" + serviceName + "/" + testName + "/failure/1.0",
          "No key in data element with id='"
             + keyData.getId()
             + "' in the metadata configuration");
    }

    Metadata metadata = new Metadata();
    metadata.setKey(key);
    Data data = new Data();
    data.setMetadataIdRef(metadata.getId());
    Message dataRequest = new Message();
    dataRequest.setMetadata(metadata);
    dataRequest.setData(data);

    try {
       getData(dataRequest);
    } catch (PerfSONARException psEx) {
       return generateResultCodeResponse(
          ECHO_SELF_TEST_NAMESPACE + serviceType + "/" + serviceName + "/" + testName + "/failure/1.0",
          psEx.getResultDescription());
    }

    return generateResultCodeResponse(
       ECHO_SELF_TEST_NAMESPACE + serviceType + "/" + serviceName + "/" + testName + "/success/1.0",
       CODE_RESPONSE_1);

    //return getData(dataRequest);
}


protected Message executeSelfTest(Message request) throws PerfSONARException
{

    Message responseMessage = new Message();

    Metadata metadata = new Metadata();
    EventType eventType = new EventType();
    eventType.setEventType(ECHO_SELF_TEST_NAMESPACE + "1.0");
    metadata.setEventType(eventType);

    Data data = new Data();
    data.setMetadataIdRef(metadata.getId());

    Message testMessage;
    testMessage = getEcho(getTestEchoRequest(ECHO_XMLDB_ACCESS_TEST_NAMESPACE));
    buildSelfTestResponseMessage(testMessage.getMetadataIterator(), data);
    buildSelfTestResponseMessage(testMessage.getDataIterator(), data);

    testMessage = getEcho(getTestEchoRequest(ECHO_XMLDB_CONTENT_TEST_NAMESPACE));
    buildSelfTestResponseMessage(testMessage.getMetadataIterator(), data);
    buildSelfTestResponseMessage(testMessage.getDataIterator(), data);

    testMessage = getEcho(getTestEchoRequest(ECHO_RANDOM_FETCH_TEST_NAMESPACE));
    buildSelfTestResponseMessage(testMessage.getMetadataIterator(), data);
    buildSelfTestResponseMessage(testMessage.getDataIterator(), data);

    responseMessage.setMetadata(metadata);
    responseMessage.setData(data);

    return responseMessage;
}


protected Message getTestEchoRequest(String eventType) throws PerfSONARException
{

    Message message = new Message();
    message.setType("EchoRequest");
    Metadata m = new Metadata();
    Data d = new Data();
    EventType et = new EventType();
    et.setEventType(eventType);
```

```
        m.setEventType(et);
        d.setMetadataIdRef(m.getId());
        message.setMetadata(m);
        message.setData(d);

        return message;
    }


    protected void buildSelfTestResponseMessage(Iterator it, Data data)
    {

        while (it.hasNext()) {
            data.setChild((Element) it.next());
        }
    }


    protected Message generateResultCodeResponse(String resultCodeId, String resultCodeDescription)
        throws PerfSONARException
    {

        Message response = new Message();

        PerfSONARException pex =
            new PerfSONARException(resultCodeId, resultCodeDescription);
        ResultCodesUtil.createResultCodeMetadata(response, pex);

        return response;

    }

} //SQLTypeMAServiceEngine
```

Apêndice B.2 – Classe do DB Module - JdbcStorageManager

```
package org.perfsonar.service.measurementArchive.SQLType;

import org.ggf.ns.nmwg.base.v2_0.Data;
import org.ggf.ns.nmwg.base.v2_0.Element;
import org.ggf.ns.nmwg.base.v2_0.Key;
import org.ggf.ns.nmwg.base.v2_0.Message;
import org.ggf.ns.nmwg.base.v2_0.Metadata;
import org.perfsonar.base.auxiliary.AuxiliaryComponentManager;
import org.perfsonar.base.auxiliary.ComponentNames;
import org.perfsonar.base.auxiliary.components.jdbc.JdbcComponent;
import org.perfsonar.base.auxiliary.components.logger.LoggerComponent;
import org.perfsonar.base.exceptions.PerfSONARException;
import org.perfsonar.service.base.storage.NMWGStorageManager;
import org.perfsonar.service.base.util.ResultCodesUtil;
import org.perfsonar.service.measurementArchive.SQLType.dao.Dao;
import org.perfsonar.service.measurementArchive.SQLType.dao.DaoProvider;
import org.perfsonar.service.measurementArchive.SQLType.transaction.FallibleTransactionCallback;
import org.perfsonar.service.measurementArchive.SQLType.transaction.FallibleTransactionTemplate;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.TransactionStatus;

import javax.SQL.DataSource;
import java.util.Collection;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;
import java.util.ServiceLoader;
import java.util.Vector;
import org.perfsonar.service.measurementArchive.SQLType.dao.OMLDao;

import org.SQLite.*;

/**
 * NMWGStorageManager based on plain JDBC.
 *
 * Delegates all fetch and store operations to event type specific DAOs. This
 * class only contains the logic to break down store and query messages into
 * event type specific DAO operations.
 *
 * JdbcStorageManager instantiates and manages the lifetime of the JDBC data
 * source (wrapped in a BoneCP connection pool) and wraps all fetch and store
 * operations in Spring transactions.
 */
public final class JdbcStorageManager implements NMWGStorageManager
{

    private static final ServiceLoader<DaoProvider> PROVIDERS = ServiceLoader.load(DaoProvider.class);

    private final LoggerComponent logger;
    private final DataSource dataSource;
    private final FallibleTransactionTemplate tx;
    private final Map<String, Dao> daos = new HashMap<String, Dao>();

    public JdbcStorageManager() throws PerfSONARException
    {
        logger = (LoggerComponent)
            AuxiliaryComponentManager.
                getInstance().
                getComponent(ComponentNames.LOGGER);
        dataSource =
            ((JdbcComponent) AuxiliaryComponentManager.
                getInstance().
                getComponent(ComponentNames.JDBC)).getDataSource();
        tx = new FallibleTransactionTemplate(new DataSourceTransactionManager(dataSource));
    }

    public JdbcStorageManager(Metadata md) throws PerfSONARException
    {
        logger = (LoggerComponent)
```

```
        AuxiliaryComponentManager.
            getInstance().
            getComponent(ComponentNames.LOGGER);
    dataSource =
        ((BoneCPJdbcComponent) AuxiliaryComponentManager.
            getInstance().
            getComponent(ComponentNames.JDBC)).getDataSource(md.getId());

    tx = new FallibleTransactionTemplate(new DataSourceTransactionManager(dataSource));
}

private synchronized Dao getDao(String eventType) throws PerfSONARException
{
    Dao dao = daos.get(eventType);
    if (dao == null) {
        dao = createDao(eventType);
        daos.put(eventType, dao);
    }
    return dao;
}

private Dao createDao(String eventType) throws PerfSONARException
{
    for (DaoProvider provider: PROVIDERS) {
        //Dao dao = provider.createDao(eventType, dataSource);
        Dao dao = new OMLDao(dataSource);
        if (dao != null) {
            return dao;
        }
    }
    throw new PerfSONARException("error.ma.query", "Unsupported event type: " + eventType);
}

@Override
public void initStorage(Key dataRepository)
    throws PerfSONARException
{
}


public Message fetch(final Metadata dataQuery)
    throws PerfSONARException
{
    if (dataQuery == null) {
        throw new PerfSONARException(
            "error.ma.query",
            "SQLStorageManager.fetch: "
                + "Query message is null");
    }

    Message response = new Message();

    String eventType = dataQuery.getEventType().getEventType();
    OMLDao dao = new OMLDao(dataSource);
    response.addChild(dao.fetch(dataQuery));

    return response;
}


@Override
public Message fetch(final Message dataQuery)
    throws PerfSONARException
{
    if (dataQuery == null) {
        throw new PerfSONARException(
            "error.ma.query",
            "SQLStorageManager.fetch: "
                + "Query message is null");
    }



    Message response = new Message();

    Collection<Metadata> collection = dataQuery.getMetadataMap().values();
```

```
    for (Metadata md : collection) {
        //Key key = md.getKey();
        String eventType = md.getEventType().getEventType();
        //Dao dao = getDao(eventType);

        OMLDao dao = new OMLDao(dataSource);


        response.addChild(md);
        //response.addChild(dao.fetch(md.getKey()));
        response.addChild(dao.fetch(md));
    }

    return response;
}


@Override
public Message store(final Message dataSet)
    throws PerfSONARException
{
    if (dataSet == null) {
        throw new PerfSONARException(
            "error.ma.query",
            "SQLStorageManager.store: Input request is null");
    }

    tx.execute(new FallibleTransactionCallback<Void, PerfSONARException>() {
        @Override
        public Void doInFallibleTransaction(TransactionStatus transactionStatus)
            throws PerfSONARException
        {
            Collection<Metadata> collection = dataSet.getMetadataMap().values();
            for (Metadata md : collection) {
                String eventType =
                    md.getKey().getParameterByName("eventType").getParameterValue();
                Dao dao = getDao(eventType);

                Data data = dataSet.getDataArray()[0];

                Vector datumVector = data.getDatum();
                if (datumVector == null) {
                    throw new PerfSONARException(
                        "error.ma.query",
                        "SQLStorageManager.store: Data element with id='" + data.getId() + "' is empty.");
                }

                for (Object datum: datumVector) {
                    dao.store(md, (Element) datum);
                }
            }
            return null;
        }
    });

    PerfSONARException pex = new PerfSONARException(
        "success.ma.write",
        "Data have been stored");
    return ResultCodesUtil.createResultCodeMetadata(new Message(), pex);
}

@Override
public Message remove(Message dataSet) throws PerfSONARException
{
    return null;
}

@Override
public void removeAll() throws PerfSONARException
{
}

@Override
public Message update(Message dataSet) throws PerfSONARException
{
```

```
        return null;
    }
}
```

Apêndice B.3 – Classe do DB Module - OMLDao

```
package org.perfsonar.service.measurementArchive.SQLType.dao;

import org.ggf.ns.nmwg.base.v2_0.Data;
import org.ggf.ns.nmwg.base.v2_0.Element;
import org.ggf.ns.nmwg.base.v2_0.Key;
import org.ggf.ns.nmwg.base.v2_0.Metadata;

import org.ggf.ns.nmwg.tools.IPERF.v2_0.Datum;
import org.perfsonar.base.exceptions.PerfSONARException;
import org.springframework.jdbc.core.namedparam.MapSQLParameterSource;
import org.springframework.jdbc.core.namedparam.SQLParameterSource;

import javax.SQL.DataSource;
import java.SQL.ResultSet;
import java.SQL.SQLException;
import java.SQL.Types;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import org.ggf.ns.nmwg.base.v2_0.Parameter;
import org.ggf.ns.nmwg.base.v2_0.Parameters;

import static org.perfsonar.service.measurementArchive.SQLType.dao.ParseUtils.parseBigDecimal;
import static org.perfsonar.service.measurementArchive.SQLType.dao.ParseUtils.parseFloat;
import static org.perfsonar.service.measurementArchive.SQLType.dao.ParseUtils.parseInteger;
import static org.perfsonar.service.measurementArchive.SQLType.dao.ParseUtils.trim;

public class OMLDao implements Dao
{
    private static final String NS_NMWG_BASE_2_0 = "http://ggf.org/ns/nmwg/base/2.0/";

    private final DaoSQLiteTemplate jdbc;

    public OMLDao(DataSource dataSource) throws PerfSONARException
    {
        jdbc = new DaoSQLiteTemplate(dataSource);
    }

    @Override
    public Data fetch(Key key)
        throws PerfSONARException
    {
        return jdbc.select(key,
                    "select * from IPERF_transfer",
                    new DataResultSetExtractor());
    }

    public Data fetch(Metadata md)
        throws PerfSONARException
    {
        String query = md.getParameters().getParameterArray().length > 0 ? "select * from fibremp where " : "select * from
fibremp";
//        String query = "select * from fibremp";
        Parameters params = md.getParameters();
        if (params != null) {
            String paramID = "";
            Parameter param = null;
            for (int i = 0; i < params.getParameterArray().length; i++) {
                paramID = params.getParameterArray()[i].getParameterName();
                param = params.getParameterByName(paramID);
                if (i+1 < params.getParameterArray().length){
                    query += paramID + " = \"" + param.getParameterValue() + "\" and ";
                }
                else {
                    query += paramID + " = " + param.getParameterValue();
                }
```

```
        }
    }
    return jdbc.select(query, new DataResultSetExtractor());
}

private static class DataResultSetExtractor extends AbstractDataResultSetExtractor
{
    @Override
    protected Element toDatum(ResultSet rs) throws SQLException
    {
        Datum datum = new Datum();
        addToDatum(datum, "Alias", rs, "Alias");
        addToDatum(datum, "DSCP", rs, "DSCP");
        addToDatum(datum, "Internval", rs, "Internval");
        addToDatum(datum, "OID", rs, "OID");
        addToDatum(datum, "SNMPCommunity", rs, "SNMPCommunity");
        addToDatum(datum, "SNMPinterVersion", rs, "SNMPVersion");
        addToDatum(datum, "Start_time", rs, "Start_time");
        addToDatum(datum, "address", rs, "address");
        addToDatum(datum, "advisoryWindowsize", rs, "advisoryWindowsize");
        addToDatum(datum, "alpha ", rs, "alpha");
        addToDatum(datum, "anonymizationLevel", rs, "anonymizationLevel");
        addToDatum(datum, "argument", rs, "argument");
        addToDatum(datum, "arguments", rs, "arguments");
        addToDatum(datum, "authRealm", rs, "authRealm");
        addToDatum(datum, "bufferLength ", rs, "bufferLength");
        addToDatum(datum, "bytes", rs, "bytes");
        addToDatum(datum, "capacity", rs, "capacity");
        addToDatum(datum, "characteristics", rs, "characteristics");
        addToDatum(datum, "classOfService", rs, "classOfService");
        addToDatum(datum, "clp", rs, "clp");
        addToDatum(datum, "command", rs, "command");
        addToDatum(datum, "count", rs, "count");
        addToDatum(datum, "dataSource", rs, "dataSource");
        addToDatum(datum, "deadline ", rs, "deadline");
        addToDatum(datum, "description", rs, "description");
        addToDatum(datum, "direction", rs, "direction");
        addToDatum(datum, "duplicates", rs, "duplicates");
        addToDatum(datum, "duration", rs, "duration");
        addToDatum(datum, "end_time", rs, "end_time");
        addToDatum(datum, "endtime", rs, "endtime");
        addToDatum(datum, "event_number", rs, "event_number");
        addToDatum(datum, "event_specification", rs, "event_specification");
        addToDatum(datum, "event_type", rs, "event_type");
        addToDatum(datum, "exponential", rs, "exponential");
        addToDatum(datum, "file", rs, "file");
        addToDatum(datum, "filter", rs, "filter");
        addToDatum(datum, "firstTtl", rs, "firstTtl");
        addToDatum(datum, "fixed", rs, "fixed");
        addToDatum(datum, "flow_h_filter", rs, "flow_h_filter");
        addToDatum(datum, "flow_p_filter", rs, "flow_p_filter");
        addToDatum(datum, "flow_s_thre", rs, "flow_s_thre");
        addToDatum(datum, "format ", rs, "format");
        addToDatum(datum, "granularity ", rs, "granularity");
        addToDatum(datum, "groupSize", rs, "groupSize");
        addToDatum(datum, "hop", rs, "hop");
        addToDatum(datum, "hospedeironame", rs, "hospedeironame");
        addToDatum(datum, "id", rs, "id");
        addToDatum(datum, "ifAddress", rs, "ifAddress");
        addToDatum(datum, "ifDescription", rs, "ifDescription");
        addToDatum(datum, "ifHospedeironame", rs, "ifHospedeironame");
        addToDatum(datum, "ifIndex", rs, "ifIndex");
        addToDatum(datum, "ifName", rs, "ifName");
        addToDatum(datum, "interface", rs, "interface");
        addToDatum(datum, "interfaceIdRef", rs, "interfaceIdRef");
        addToDatum(datum, "interval", rs, "interval");
        addToDatum(datum, "interval_sec", rs, "interval_sec");
        addToDatum(datum, "interval_usec ", rs, "interval_usec");
        addToDatum(datum, "iqrIpd", rs, "iqrIpd");
        addToDatum(datum, "jitter", rs, "jitter");
        addToDatum(datum, "latest", rs, "latest");
        addToDatum(datum, "linkIdRef", rs, "linkIdRef");
        addToDatum(datum, "loss", rs, "loss");
        addToDatum(datum, "lossPercent", rs, "lossPercent");
```

```
            addToDatum(datum, "loss_rate", rs, "loss_rate");
            addToDatum(datum, "mask", rs, "mask");
            addToDatum(datum, "maxIpd", rs, "maxIpd");
            addToDatum(datum, "maxRtt", rs, "maxRtt");
            addToDatum(datum, "maxTtl", rs, "maxTtl");
            addToDatum(datum, "mbps", rs, "mbps");
            addToDatum(datum, "meanIpd", rs, "meanIpd");
            addToDatum(datum, "meanRtt", rs, "meanRtt");
            addToDatum(datum, "measurementOffset", rs, "measurementOffset");
            addToDatum(datum, "measurementPeriod", rs, "measurementPeriod");
            addToDatum(datum, "medianRtt", rs, "medianRtt");
            addToDatum(datum, "messageIdRef", rs, "messageIdRef");
            addToDatum(datum, "metadataidRef", rs, "metadataidRef");
            addToDatum(datum, "minIpd", rs, "minIpd");
            addToDatum(datum, "minRtt", rs, "minRtt");
            addToDatum(datum, "name", rs, "name");
            addToDatum(datum, "netflowversion", rs, "netflowversion");
            addToDatum(datum, "networkIdRef", rs, "networkIdRef");
            addToDatum(datum, "no_capt", rs, "no_capt");
            addToDatum(datum, "no_lost", rs, "no_lost");
            addToDatum(datum, "numBytes", rs, "numBytes");
            addToDatum(datum, "numBytesUnits", rs, "numBytesUnits");
            addToDatum(datum, "numberOfTests", rs, "numberOfTests");
            addToDatum(datum, "outOforder", rs, "outOforder");
            addToDatum(datum, "packetInterval", rs, "packetInterval");
            addToDatum(datum, "packetSize", rs, "packetSize");
            addToDatum(datum, "packetSize", rs, "packetSize");
            addToDatum(datum, "packetTTL", rs, "packetTTL");
            addToDatum(datum, "packets", rs, "packets");
            addToDatum(datum, "pathIdRef", rs, "pathIdRef");
            addToDatum(datum, "pause", rs, "pause");
            addToDatum(datum, "pkts", rs, "pkts");
            addToDatum(datum, "port", rs, "port");
            addToDatum(datum, "precedence", rs, "precedence");
            addToDatum(datum, "precision", rs, "precision");
            addToDatum(datum, "precisionUnits", rs, "precisionUnits");
            addToDatum(datum, "protocol ", rs, "protocol");
            addToDatum(datum, "protocols", rs, "protocols");
            addToDatum(datum, "queryNum ", rs, "queryNum");
            addToDatum(datum, "role", rs, "role");
            addToDatum(datum, "samplerate", rs, "samplerate");
            addToDatum(datum, "scheduleInterval", rs, "scheduleInterval");
            addToDatum(datum, "sent", rs, "sent");
            addToDatum(datum, "seqNum ", rs, "seqNum");
            addToDatum(datum, "sequencenumber", rs, "sequencenumber");
            addToDatum(datum, "size", rs, "size");
            addToDatum(datum, "start_time", rs, "start_time");
            addToDatum(datum, "starttime", rs, "starttime");
            addToDatum(datum, "statistics", rs, "statistics");
            addToDatum(datum, "subnet", rs, "subnet");
            addToDatum(datum, "supportedEventType", rs, "supportedEventType");
            addToDatum(datum, "syncronized", rs, "syncronized");
            addToDatum(datum, "table", rs, "table");
            addToDatum(datum, "timeDuration ", rs, "timeDuration");
            addToDatum(datum, "timeType", rs, "timeType");
            addToDatum(datum, "timeValue", rs, "timeValue");
            addToDatum(datum, "timeout", rs, "timeout");
            addToDatum(datum, "timestamp", rs, "timestamp");
            addToDatum(datum, "transportProtocol", rs, "transportProtocol");
            addToDatum(datum, "ttl", rs, "ttl");
            addToDatum(datum, "type", rs, "type");
            addToDatum(datum, "value", rs, "value");
            addToDatum(datum, "value ", rs, "value");
            addToDatum(datum, "valueUnits", rs, "valueUnits");
            addToDatum(datum, "waitTime", rs, "waitTime");
            addToDatum(datum, "windowSize ", rs, "windowSize");
            addToDatum(datum, "windowSizeRequest", rs, "windowSizeRequest");
            addToDatum(datum, "throughput", rs, "throughput");
            return datum;
        }
    }
}
```

Apêndice C – Configuração para compilação do IPERF

```
defApplication('oml:app:IPERF', 'IPERF') do |a|

  a.path = "@bindir@/IPERF"
  a.version(2, 9, -0) # This is the version of the OML instrumentation, not the base IPERF
  a.shortDescription = 'IPERF traffic generator and bandwidth measurement tool'
  a.description = %{
IPERF is a traffic generator and bandwidth measurement tool. It provides
generators producing various forms of packet streams and port for sending these
packets via various transports, such as TCP and UDP.
  }

  a.defProperty('interval', 'pause n seconds between periodic bandwidth reports',
                '-i', {:type => :double, :dynamic => false})
  a.defProperty('len', 'set length read/write buffer to n (default 8 KB)',
                '-l', {:type => :integer, :dynamic => false})
  a.defProperty('print_mss', 'print TCP maximum segment size (MTU - TCP/IP header)',
                '-m', {:type => :boolean, :dynamic => false})
  a.defProperty('output', 'output the report or error message to this specified file',
                '-o', {:type => :string, :dynamic => false})
  a.defProperty('port', 'set server port to listen on/connect to to n (default 5001)',
                '-p', {:type => :integer, :dynamic => false})
  a.defProperty('udp', 'use UDP rather than TCP',
                '-u', {:order => 2, :type => :boolean, :dynamic => false})
  a.defProperty('window', 'TCP window size (socket buffer size)',
                '-w', {:type => :integer, :dynamic => false})
  a.defProperty('bind', 'bind to <hospedeiro>, an interface or multicast address',
                '-B', {:type => :string, :dynamic => false})
  a.defProperty('compatibility', 'for use with older versions does not sent extra msgs',
                '-C', {:type => :boolean, :dynamic => false})
  a.defProperty('mss', 'set TCP maximum segment size (MTU - 40 bytes)',
                '-M', {:type => :integer, :dynamic => false})
  a.defProperty('nodelay', 'set TCP no delay, disabling Nagle\'s Algorithm',
                '-N', {:type => :boolean, :dynamic => false})
  a.defProperty('IPv6Version', 'set the domain to IPv6',
                '-V', {:type => :boolean, :dynamic => false})
  a.defProperty('reportexclude', '[CDMSV]   exclude C(connection) D(data) M(multicast) S(settings) V(server) reports',
                '-x', {:type => :string, :dynamic => false})
  a.defProperty('reportstyle', 'C or c for CSV report, O or o for OML',
                '-y', {:type => :string, :dynamic => false})

  a.defProperty('server', 'run in server mode',
                '-s', {:type => :boolean, :dynamic => false})

  a.defProperty('bandwidth', 'set target bandwidth to n bits/sec (default 1 Mbit/sec)',
                '-b', {:type => :string, :dynamic => false})
  a.defProperty('client', 'run in client mode, connecting to <hospedeiro>',
                '-c', {:order => 1, :type => :string, :dynamic => false})
  a.defProperty('dualtest', 'do a bidirectional test simultaneously',
                '-d', {:type => :boolean, :dynamic => false})
  a.defProperty('num', 'number of bytes to transmit (instead of -t)',
                '-n', {:type => :integer, :dynamic => false})
  a.defProperty('tradeoff', 'do a bidirectional test individually',
                '-r', {:type => :boolean, :dynamic => false})
  a.defProperty('time', 'time in seconds to transmit for (default 10 secs)',
                '-t', {:type => :integer, :dynamic => false})
  a.defProperty('fileinput', 'input the data to be transmitted from a file',
                '-F', {:type => :string, :dynamic => false})
  a.defProperty('stdin', 'input the data to be transmitted from stdin',
                '-I', {:type => :boolean, :dynamic => false})
  a.defProperty('listenport', 'port to recieve bidirectional tests back on',
                '-L', {:type => :integer, :dynamic => false})
  a.defProperty('parallel', 'number of parallel client threads to run',
                '-P', {:type => :integer, :dynamic => false})
  a.defProperty('ttl', 'time-to-live, for multicast (default 1)',
                '-T', {:type => :integer, :dynamic => false})
  a.defProperty('linux-congestion', 'set TCP congestion control algorithm (Linux only)',
                '-Z', {:type => :boolean, :dynamic => false})

  a.defMeasurement("application"){ |m|
```

```
    m.defMetric('pid', :integer, 'Main process identifier')
    m.defMetric('version', :string, 'IPERF version')
    m.defMetric('cmdline', :string, 'IPERF invocation command line')
    m.defMetric('starttime_s', :integer, 'Time the application was received (s)')
    m.defMetric('starttime_us', :integer, 'Time the application was received (us)')
  }

  a.defMeasurement("settings"){ |m|
    m.defMetric('pid', :integer, 'Main process identifier')
    m.defMetric('server_mode', :integer, '1 if in server mode, 0 otherwise')
    m.defMetric('bind_address', :string, 'Address to bind')
    m.defMetric('multicast', :integer, '1 if listening to a Multicast group')
    m.defMetric('multicast_ttl', :integer, 'Multicast TTL if relevant')
    m.defMetric('transport_protocol', :integer, 'Transport protocol (IANA number)')
    m.defMetric('window_size', :integer, 'TCP window size')
    m.defMetric('buffer_size', :integer, 'UDP buffer size')
  }

  a.defMeasurement("connection"){ |m|
    m.defMetric('pid', :integer, 'Main process identifier')
    m.defMetric('connection_id', :integer, 'Connection identifier (socket)')
    m.defMetric('local_address', :string, 'Local network address')
    m.defMetric('local_port', :integer, 'Local port')
    m.defMetric('remote_address', :string, 'Remote network address')
    m.defMetric('remote_port', :integer, 'Remote port')
  }

  a.defMeasurement("fibremp"){ |m|
    m.defMetric('pid', :integer, 'Main process identifier')
    m.defMetric('connection_id', :integer, 'Connection identifier (socket)')
    m.defMetric('begin_interval', :float, 'Start of the averaging interval (IPERF timestamp)')
    m.defMetric('end_interval', :float, 'End of the averaging interval (IPERF timestamp)')
    m.defMetric('size', :uint64, 'Amount of transmitted data [Bytes]')
  }

  a.defMeasurement("losses"){ |m|
    m.defMetric('pid', :integer, 'Main process identifier')
    m.defMetric('connection_id', :integer, 'Connection identifier (socket)')
    m.defMetric('begin_interval', :float, 'Start of the averaging interval (IPERF timestamp)')
    m.defMetric('end_interval', :float, 'End of the averaging interval (IPERF timestamp)')
    m.defMetric('total_datagrams', :integer, 'Total number of datagrams')
    m.defMetric('lost_datagrams', :integer, 'Number of lost datagrams')
  }

  a.defMeasurement("jitter"){ |m|
    m.defMetric('pid', :integer, 'Main process identifier')
    m.defMetric('connection_id', :integer, 'Connection identifier (socket)')
    m.defMetric('begin_interval', :float, 'Start of the averaging interval (IPERF timestamp)')
    m.defMetric('end_interval', :float, 'End of the averaging interval (IPERF timestamp)')
    m.defMetric('jitter', :float, 'Average jitter [ms]')
  }

  a.defMeasurement("packets"){ |m|
    m.defMetric('pid', :integer, 'Main process identifier')
    m.defMetric('connection_id', :integer, 'Connection identifier (socket)')
    m.defMetric('packet_id', :integer, 'Packet sequence number for datagram-oriented protocols')
    m.defMetric('packet_size', :integer, 'Packet size')
    m.defMetric('packet_time_s', :integer, 'Time the packet was processed (s)')
    m.defMetric('packet_time_us', :integer, 'Time the packet was processed (us)')
    m.defMetric('packet_sent_time_s', :integer, 'Time the packet was sent (s) for datagram-oriented protocols')
    m.defMetric('packet_sent_time_us', :integer, 'Time the packet was sent (us) for datagram-oriented protocols')
  }


end

# Local Variables:
# mode:ruby
# vim: ft=ruby:sw=2
# End:
```