ORBIT Measurements Framework and Library (OML): Motivations, Design, Implementation, and Features

Manpreet Singh, Maximilian Ott, Ivan Seskar, Pandurang Kamat
WINLAB, Rutgers University, 73 Brett Road, Piscataway, NJ 08854
{singh, max, Seskar, pkamat}@winlab.rutgers.edu

Abstract

In this paper we present ORBIT measurement framework and library (OML), which is a distributed software framework enabling real-time collection of data in a large distributed environment. The success of a multi-user distributed testbed facility depends largely on the ease of use, remote access as well as on the ease of collecting useful measurements from experimental runs. OML provides a flexible and dynamic way in which data is collected and made available for real-time access to the experimenters. Application programmers can use simple interfaces provided to transfer measurements and other performance data to a central repository. This paper focuses on the motivation, requirements, design, implementation and real world usage of OML that is designed to provide a scalable, controllable and easy to use mechanism for experimenters to collect useful results from the experiments conducted on the ORBIT testbed [1].

1. Introduction

One of key challenges faced by an experimenter using a distributed large-scale testbed is how to collect experiment data efficiently. Traditionally, the measurement data are locally written into log files and are collected at the end of the experiment. A large collection of nodes and huge amount of measurement data generated during the experiment pertaining to node, network and application performance, results in a number of logging files in various formats. Additionally many of the experiment parameters, such as input parameters, may not be captured at all. Another problem with logging files is that they require some form of data serialization to a text file and back for analysis making analysis across multiple applications difficult. Also, the current data collection mechanisms create excessive overhead, especially in the maintenance of experiment results for future use.

It is important to have a scalable, easy to use, distributed and controllable framework to collect and organize experiment data, and analyze the results in real-time. A significant advantage of real-time data collection is that it allows for interactive experiments in which users can react to the dynamics of the experiment immediately, saving valuable resources. It can reduce the burden of measurement collection on the experimenters so that they can focus on protocol and application development without worrying about the complexity and details to collect, transport and store the experiment data.

We propose OML, which is a measurement data collection and organization framework that addresses the above challenges. It enables the experimenter to define the measurement points and parameters, collect and pre-process measurements, and organize the collected data into a single database with the experiment's context, avoiding logging files in various formats. The OML framework is based on a client/server architecture and uses IP multicast for the client to report the collected data to the server in real-time. It defines the data structures and functions for sending/receiving, encoding/decoding and storing experiment data. With user-friendly and generic APIs, it can be easily integrated into user applications. Users can define what measurements are to be collected and stored. The clients at the experiment nodes collect measurements and send them to the collection server over a multicast channel after encoding them into XDR [2] format. OML supports multiple multicast channels and instances of the collection server per experiment to enhance the network scalability and provide reliability of data collection by load balancing and redundancy. An SQL database is used for persistent storage of experiment data that also allows access using standard data analysis tools like Matlab [6]. Note that although OML is written initially with a focus on the ORBIT testbed [1], it can be used in various wired and wireless

networking testbeds and distributed systems for data collection.

The rest of this paper is organized as follows. Section 2 discusses the requirements for data collection posed by distributed large-scale network testbeds and the key challenges in building such a collection framework. In Section 3, the OML architecture and implementation details are described with reference to the requirements and features. It also discusses the APIs provided to interface with application code and the methods to control the collection behavior. Section 4 presents the performance of our implementation as well as the experience gained through OML usage on the ORBIT testbed [1]. Finally Section 5 concludes the paper.

2. Requirements posed by a distributed framework.

The initial goal of OML is to provide a mechanism for large-scale testbed users the ability to transfer their measurements into a database on a remote machine. Traditionally, if the application is running on a number of nodes, after the experiment concludes, users have to log-in into all the machines and manually copy the measurement files and system logs to a remote machine for further analysis. This is a time consuming and repetitive process, which delays the execution of the next set of experiments waiting for the resources to become available. It may also result in missing files.

Further, if the experimenter wishes to change the collection behavior, he/she needs to recompile and re-deploy the application, which itself is an error-prone and time consuming process. Hence a framework, which simplifies the data collection process, scales with the size of the distributed system and allows dynamic control over the measurement collection process, is required in such a distributed networking environment.

The motivation behind OML is to hide the complexity of data collection from the experimenters so that they can focus on application development and logic. Principle requirements of a data collection framework in a distributed environment include

- **User friendly:**

Provide simple and user friendly APIs for the application developer to collect and transport the experiment data. This includes handling any threading issues related to data collection, data-type safety and minimal configuration and instantiation complexity on the part of the application developer.

- **Controllability and Management**

It is time-consuming and complex to re-write, recompile and re-deploy the application each time one wants to change the collection behavior. There should be a simple way to control and change this behavior in real-time.

- **Accountability**

Framework should provide a way to correlate application measurements and related data, in time (e.g. timestamp) and context (e.g. sequence numbers, name of the machine running the application and other hardware/software characteristics).

- **Collocation of information**

Traditionally, in a large distributed environment, all the information related to the experiment is not available at a central point, making it difficult to correlate events in an experiment with its configuration options and other variable parameters associated with the execution environment. The collection framework should provide a central point where experimenters can look for data related to the distributed environment in which experiments are run.

- **Scalability**

The framework should not introduce network traffic large enough to have a detrimental effect on the regular application/control performance. It should make sure that the processing load caused by the collection framework on the machines running the application is minimal.

- **Flexible and Generic Solution**

The collection framework should be generic enough that it can be used to collect not only application measurement's data; but also any other data like system and network statistics, application parameter and debug logging etc.

3. OML Architecture and Implementation.

OML aims at reducing the burden of measurements collection on application developers. It defines the framework, data structures and functions for transporting and storing experiment data. Data filters form another sub-component of this library that allows testbed users to compress/reduce the measurements by applying various averaging, linear and non-linear algorithms. From an operational perspective OML is based on a client server paradigm, where clients are the nodes running application code that dispatch the measurements; and the server is a machine that receives, decodes and stores this data in the SQL database.

Figure 1 shows the high level architecture of OML, with client side and server side components communicating through IP multicast. If the environment consists of a large number of nodes generating massive amounts of measurement traffic,

multiple multicast channels can be used in conjunction with virtual lans (vlans) to distribute the network load. Multiple collection servers may subscribe to the same multicast address to provide runtime redundancy to the collection mechanism. Thus using multicast in OML serves to improve both scalability and reliability of the collection framework.

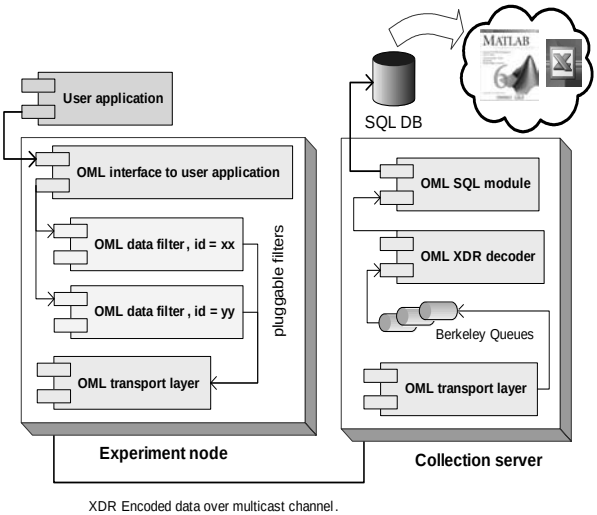

Figure 1. OML component architecture

3.1. Client side components

3.1.1. API interface. This interface provides user applications with the ability to transport collection data through the OML framework. It also provides a type safe way of transferring data over the network and handling the threading issues if any.

3.1.2 OML data filters. These are pluggable components that provide a standard way of reducing the amount of collectable data to be stored for further analysis. More the amount of data we capture, the more we have to transport and store; hence exhausting the disk and network resources. On the other hand, filtering too aggressively might "throw away" details which turn out to be crucial in understanding certain phenomena, resulting in re-run of the experiment with different filter settings.

Filters can be configured and used without re-writing the application code and hence provide a flexible and efficient way to change the data collection behavior. OML supports time triggered filtering, where filters are fired after certain amount of time; and sample triggered filtering in which case filters are fired based on the number of data values collected.

3.1.3 OML XDR Encoding and transport layer. This module is responsible for encoding the filtered measurements data into XDR format and sending it to the OML server over a multicast channel. Each encoded packet corresponds to a measurement point and contains its name thus helping the server to identify the measurement point the packet belongs to. This module provides a memory and network efficient way of transferring experiment data.

3.2. Server side components.

3.2.1. Berkeley database queue (bdb queue). SleepyCat's Berkeley database [3], which is an embedded database that supports key based fast access persistent queues, is used to store the received packets. The logic behind such a design is the fact that XDR decoding and SQL insert process is much slower than the data receiving process. Using a queue significantly improves the scalability of OML by providing a buffer to avoid packet loss when dealing with experiments that generate bursty data. Since bdb queues are used as a pluggable component, OML transport layer can feed into multiple bdb queues to accommodate data load dynamically.

3.2.2. XDR Decoder. Decoder reads out of the bdb queue and decodes the XDR packet according to the server configuration file. Both, the client and the server configuration files are generated from the same application and experiment definition files; this ensures that decoding is done in a type safe manner.

3.2.3. SQL Module. This module is responsible for storing the decoded values in the SQL server for post experiment analysis and data persistency. Since each OML packet contains the name of the measurement point, which in turn is mapped to a unique database table; it is used to identify the correct table where the measurement values are to be stored. OML currently uses MySQL server [4], but any SQL compliant database is supported. Popular data analysis tools like Matlab and Microsoft Excel can directly import data from an SQL database, hence significantly enhancing the usability of OML.

3.3. OML configuration and setup

3.3.1 Code generation for OML client API. Client API provides clean interfaces for the application developers making it easy for the users to integrate measurement collection capabilities into their applications. Application developers also don't have to worry about the threading issues as they are handled by the OML.

An application developer can define the measurement points and parameters for his/her application through a web interface. As shown in Figure 2, the definition is saved into an XML-based configuration file. Based on the definition, the source code for the measurement client is automatically generated by an XSLT based code generator. At the client side, this automatically generated code contains application specific methods that handle type safe data collection, which can be compiled and linked with the application.

```
<measurement-points>
  <measurement-point id="group1">
    <metric id="rssi" type="float"/>
    <metric id="noise" type="float"/>
  </measurement-point>
  <measurement-point id="group2">
    <metric id="lost_packets" type="float"/>
  </measurement-point>
<measurement-points>
```

Code Generation

```
int oml_group1 (float rssi,
                float noise
                ) {…}

int oml_group2 (float lost_packets
                ) {…}
```

Compile Application code against Client APIs

OML Ready Application

Figure 2. Generating client APIs

Figure 2 shows application definition containing radio parameters (rssi, noise and throughput) that a user wants to collect. The XML definition file shows two measurement points, "*group1*" & "*group2*" defined by the application programmer. Based on the definition, the source code is automatically generated with the API functions *oml_group1 (...)* and *oml_group2 (...)*.

The application then calls the measurement point APIs to transport the measurements data to the collection server. OML handles the threading issues involved with the data filtering, encoding and transmission. Following the example of application definition shown in Figure 2, the OML API calls from the application are shown in Figure 3.

```
if(r_data->send_option == 1) {
        buffer->rssi = recv_packet_params.rssi;
        buffer->noise = recv_packet_params.noise;

        oml_group1(buffer->rssi, buffer->noise);
} else {
    syslog(LOG_ERR, "Unknown receive option!! \n");
}

lost_packets = (int) (pck_id.seqnum - old - 1);
oml_group2(lost_packets);
```

Figure 3. Calling OML API from application code

3.3.2 OML data filter configuration. Filter configuration is done as a part of experiment definition. As shown by a snippet of sample experiment definition in figure 4, a filter "*example_filter*" is chosen to be applied on measured rssi values, and fired using a "time trigger". The experiment definition file also defines a *trigger* property for the measurement point. The *value* element of this property determines when all the filters included in the measurement point get triggered. The *refid* attribute of filter element gives the name of the filter, and the properties specify any required filter parameters that are need for its operation.

```
<measurement-point refid="group2" type="time_triggered">
   <properties>
     <property name="trigger">
       <value units="sec">5</value>
     </property>
   </properties>
   <metric name="lost_packets">
     <filter refid="example_filter">
          <properties>
            <property name="param1" value="10.5"/>
          </properties>
     </filter>
   </metric>
</measurement-point>
```

Figure 4. Filter configuration using experiment definition

An experimenter can either use one of the default filters or write a custom filter using the APIs provided by OML and integrate it with the framework. A base filter class *OMLFilter* is provided as part of OML. A custom filter class must be derived from this base class and the function *get_filtered_values* overridden. In addition to this, the filter definition, conforming to the OML filter schema, should be provided in XML format. This definition should list the input and output

parameters of the filter along with their data types. Sample code for a simple filter is shown in figure 5.

```
class example_filter: public OMLFilter{
  int filter_param1;

  example_filter (Hashtable filter_params)
  {        …        }

  vector <void *> get_filtered_values (
                    vector  <void *> measurement_values
                    int value _data_type
                  )
  {        …        }
};
```

Figure 5. Data filter API

It requires, as input, the measurement values that need to be processed and the data type of the values (0, 1, 2 for integer, float, long respectively) and returns a void pointer to the results. Filters are applied per metric in a measurement point. Filter parameters are passed using a hash table in the filter constructor. These filter parameters are derived from the experiment definition, as shown in Figure 4.

3.3.3 Client Side Operation. As and when a set of measurement values are available, the application calls OML client API functions such as *oml_group1* and *oml_group2* to pass these values to "measurement points".

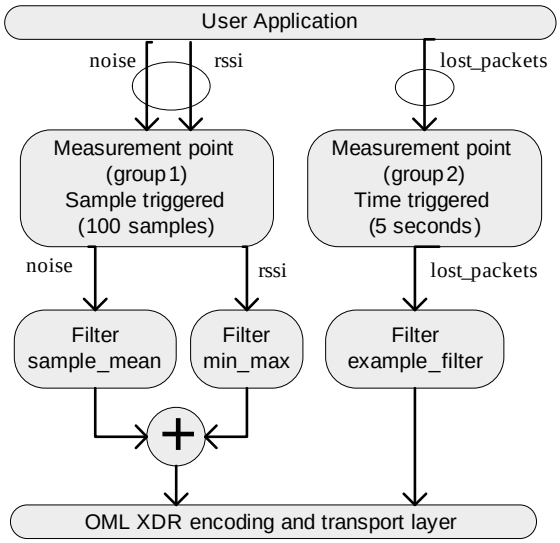

Figure 6. Client side measurement data flow

A "measurement point" accumulates all the incoming values until the trigger condition (time or sample based) is met, in which case the "measurement point" fires all the filters associated with all the metrics. The results are then combined into one outbound message and sent to the XDR encoding layer, which eventually multicasts the encoded values to the collection server. As seen in Figure 6, metrics can be filtered using various filter types, by associating with different measurement points.

3.3.4 Database configuration. At the collection server side, the application definition is used to create database schema for the experiment. OML uses XSLT to convert the application definition to a database schema file.

As shown in Figure 7, a database table is created corresponding to each measurement point; and the table names are derived from the *id* attribute of the *group* element, i.e. the names of the measurement points. Each table has sequence number, timestamp and the OML client's name/id as mandatory fields; in addition to the columns which correspond to the *id* attribute of each "metric" element. Once the testbed user defines the experiment, the application definition is used in conjunction with the experiment definition to create OML client and server configuration files.

```
<measurement-points>
   <measurement-point  id="group1">
         <metric id="rssi" type="float"/>
         <metric id="noise" type="float"/>
   </measurement-point>
</measurements-points>
```

```
mysql> describe group1;
+-------------+---------------+------+-----+---------+
| Field       | Type          | Null | Key | Default |
+-------------+---------------+------+-----+---------+
| node_id     | varchar(32)   | YES  |     | NULL    |
| sequence_no | int(11)       | YES  |     | NULL    |
| timestamp   | timestamp(14) | YES  |     | NULL    |
| rssi        | float         | YES  |     | NULL    |
| noise       | float         | YES  |     | NULL    |
+-------------+---------------+------+-----+---------+
```

Figure 7. Database schema generation

4. Deployment and evaluation experience

This section talks about the real-world OML usage in ORBIT [1], which is a distributed wireless testbed. The ease of collecting and analyzing data, real-time experiment control and performance analysis is discussed.

4.1. Example experimental setup

A traffic generator application was written to get the rssi (received signal strength) for each packet, in addition to the offered load values for the senders, and throughput values for the receivers. OML interface was

used to input the information about the measurement points leading to the generation of an application definition file. This file served as an input to the XSLT based code generator to automatically generate the client API, which in turn was integrated with the application code. The application definition file was also used to generate the database schema.

In the second step, the user defined the experiment by choosing the data filters for each measurement point defined in the application definition. This experiment definition was used in conjunction with the application definition to generate configuration files for the client nodes and the OML server. Both, the application and the experiment definition were stored in the database with the experiment results.

Four runs of the same experiment were done by simply changing the filter parameters to gradually increase the amount of OML data generated by the experiment nodes running the application. Each time the filter parameters were changed, only the experiment definition was modified, hence avoiding the re-compilation and redeployment of the application code on the experiment nodes.

4.2. Real-time data availability and control.

The experimenter wrote a simple Perl script, shown in Figure 8, to keep track of the number of packets loss, one of the measurement data metrics reported by the application using OML. User kept increasing the data rate till the number of packets lost went beyond a threshold of 150, when the user stopped the experiment. This shows the controllability which is achievable by real-time data collection using OML framework.

```
#!/usr/bin/perl
use Mysql;
...
$dbh = Mysql->connect($hostname, $database, $user, $password);
$sql_query="select lost_packets from group 2 where node_id='node3-4' order by
sequence_no desc limit 1";

for(;;) {
   sleep(1);
   $sth = $dbh->query($sql_query);
   while(@record = $sth->FetchRow) {
      print  "$record[0]\n";
      if ($record[0] == 150) {
         quitExperiment ();
      } else {
         increaseDataRate (10);
      }
   }
}
```

Figure 8. Real-time data analysis and application control

4.3. Collocation of Information and Ease of Data Analysis

All the information pertaining to a particular experiment is stored in the database along with the experimental results. The application definition that defines what is being measured, the experiment definition that defines how it is being measured, the experiment results and the OML performance metrics are all available to the user at a single point. This allows quicker analysis and correlation of experiment results; as well as quick and easy repeatability of the same experiment. It also enables dynamic controllability of experiment by providing near real-time access to the data.

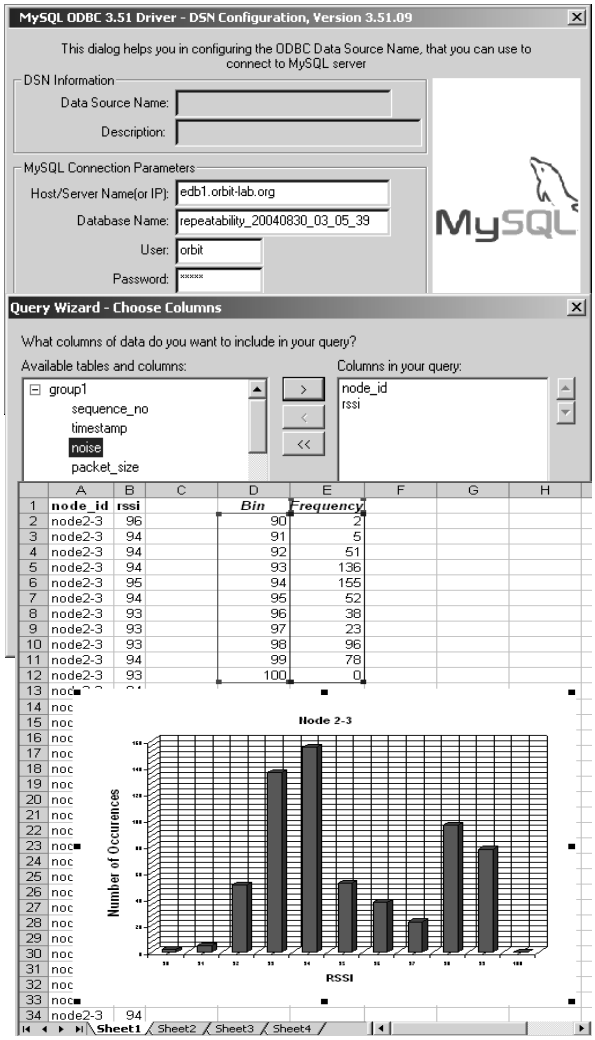

Figure 9. Import data from MySQL into Excel

Storing experimental and OML performance results in the SQL database allows the use of standard analysis tools like Matlab and Excel. Importing data into these tools is an easy and user friendly process.

Figure 9 shows the ease with which experimental results can be imported and plotted in Microsoft Excel.

First the data source is selected and then the fields to be viewed and analyzed are imported in the Excel sheet.

4.4. OML performance measurements.

OML uses itself to collect the measurements data pertaining to its own performance. The OML server collects various statistics like the number of packets received; packets dropped, XDR decoding errors, SQL errors and the bdb queue size, and store this data along with the experiment data.

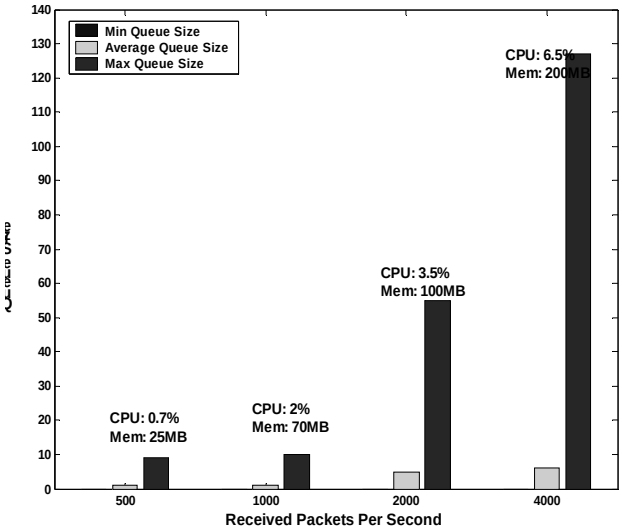

Figure 10. OML performance analyses

Figure 10 shows the performance of the OML server, which was running on a dual Xeon processor with 1 GB of memory and a gigabit network card. Performance analysis is done as a function of measurements traffic load. These results represent the average packet rate, for four different data filter configurations, from eight client nodes reporting measurements using OML. As we can see the average queue size remains small, even though the maximum queue size can be quite large due to bursty measurements traffic. No OML packet loss, XDR decoding errors and SQL errors were found.

5. Future work

The current version of OML does not allow changing filter configurations during the execution of the experiment. In future versions, we plan to support this feature as well as extend the library of data filters to provide more functionality for the same. Extensions with built-in measurements, like Ganglia [5], are also in the future roadmap. We are hopeful of deploying this framework on larger, distributed and diverse environments to further study its performance and enhance its usability.

6. Conclusion

This paper presents a generic, scalable and flexible framework for the collection of application generated of data in a distributed environment. This framework reduces the burden of data collection on application developers by providing simple APIs for transport of data in a reliable manner. Usability of the framework is significantly enhanced by use of technologies like SQL, hence allowing the use of standard tools for data analysis. Use of multicasting and Berkeley database enables a reliable and flexible framework; and provides network and computational scalability. The results show the benefits, usability and the performance of the framework.

The OML framework has been successfully deployed as part of the ORBIT testbed and has been in extensicve use over the last few months. Besides measuring experimental data, OML is being used for data collection from a third-party wireless network monitoring tool. The ORBIT [1] hardware monitoring system also uses OML to collect and report various health parameters associated with the testbed nodes.

7. References

[1] I. S. D. Raychaudhuri, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, M. Singh, "Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols," submitted to the IEEE Wireless Communications and Networking Conference, New Orleans.

[2] "RFC 1014 - XDR: External Data Representation Standard," http://www.faqs.org/rfcs/rfc1014.html

[3] Sleepycat software Berkeley DB product website, http://sleepycat.com/products/db.shtml

[4] MySQL product website, http://www.mysql.com

[5] Matthew L. Massie, Brent N. Chun, David E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience"

[6] Matlab product website, http://www.mathworks.com