An Instrumentation Framework for the Critical Task of Measurement Collection in the Future Internet

Olivier Mehani,* Guillaume Jourjon, Thierry Rakotoarivelo, and Max Ott

Abstract: Experimental research on future Internet technologies involves observing multiple metrics at various distributed points of the networks under study. Collecting these measurements is often a tedious, repetitive and error prone task, be it in a testbed or in an uncontrolled field experiment. The relevant experimental data is usually scattered across multiple hosts in potentially different formats, and sometimes buried amongst a trove of other measurements, irrelevant to the current study. Collecting, selecting and formatting the useful measurements is a time-consuming and error-prone manual operation.

In this paper, we present a conceptual Software-Defined Measurement (SDM) framework to facilitate this task. It includes a common representation for any type of experimental data, as well as the elements to process and collect the measurement samples and their associated metadata. We then present an implementation of this concept, which we built as a major extension and refactoring of the existing Orbit Measurement Library (OML). We outline its API, and how it can be used to instrument an experiment in only a few lines of code. We also evaluate the current implementation, and demonstrate that it efficiently allows measurement collection without interfering with the systems under observation.

Keywords: Network measurements, Instrumentation system, Software-defined Measurement, Performance evaluation, OML

*Corresponding author: olivier.mehani@nicta.com.au
Nicta, Sydney. Eveleigh, NSW, Australia, first.last@nicta.com.au

Published: 15 January 2014, Computer Networks, Special Issue on Future Internet Testbeds
Cite as: O. Mehani, G. Jourjon, T. Rakotoarivelo, and M. Ott. “An Instrumentation Framework for the Critical Task of Measurement Collection in the Future Internet”. In: *Computer Networks* (2014). Ed. by J. P. G. Sterbenz et al. In press

ISSN: 1833-9646-6065

1 Introduction

In his seminal 2004 paper [36], Paxson presented several good-practice guidelines on the collection, annotation and storage of experimental data and identified their critical role in sound Internet experiments. Since multiple categories of observations are collected using different tools with varying accuracy, precision, and disruption characteristics, the creation of good quality data under these guidelines often requires tedious manual and *ad hoc* post-processing. Moreover, as these tools rarely share common storage formats, further non-negligible data manipulation tasks are required before any analysis, or later reuse in other scientific studies, can be done.

The problem of collecting and reporting data becomes paramount for networking experiments and distributed applications, as they usually involve multiple entities. Furthermore, as research in future Internet technologies moves towards Software-Defined Networking (SDN) [34] and everything-as-a-service [31], such data collection processes needs to operate in highly dynamic environments where experimental parameters and performance metrics, as well as their sources and destinations—in essence the full reporting chain—may change rapidly. Yet, the observation of these environments often relies on special-purpose tools with often limited flexibility. Nonetheless, some recent works have demonstrated the value of an ability to correlate and analyse different types of measurements from various sources in, *e.g.*, investigating the root cause of observed network issues [24, 46].

Additionally, sampling and reporting have a cost. As future Internet systems become increasingly more mobile and heterogeneous, the collection of measurement data might require significant network resources at a given time. While various workaround solutions are possible, such as local buffering of measurements, adjustment of their sampling rates, or aggregation of multiple samples into summary statistics (*e.g.*, [53]), they require some decision process on the measurement probe side to adequately adapt to the currently observed conditions of the collection chain.

A common framework for data collection and reporting is clearly needed to support measurement in future Internet research. Such a framework would provide a generic foundation for *Software-Defined Measurements* (SDM). Moreover, we believe it should cater for the following requirements, which extends the good-practice guidelines from Paxson's paper [36].

(Req. 1) known, ideally minimal, systematic bias in the data collection (*i.e.*, no side effect introduced by the measurement);

(Req. 2) robust timestamping across nodes (*i.e.*, time comparisons between samples from different sources should be meaningful);

(Req. 3) rich and documented storage format (*i.e.*, supporting the best practices of [36]);

(Req. 4) ability to integrate legacy measurement tools and formats (*i.e.*, leverage existing and well-understood utilities rather than force a clean slate);

(Req. 5) domain-agnostic support (*i.e.*, not only network or system metrics);

(Req. 6) usability in distributed systems (*i.e.*, data coming from multiple points can be processed, and/or stored, on many others);

(Req. 7) reusable measurements (*i.e.*, samples collected for one purpose should be accessible for other analyses without a second observation);

(Req. 8) dynamically reconfigurable (*i.e.*, feedback loops should allow the reporting chain to adjust its own parameters based on currently observed conditions).

The Orbit Measurement Library (OML) [48] was released in 2005 for the sole purpose of instrumenting wireless experiments and reporting measurements into a single central database. We took over the development of OML from its original team and have evolved it into a generic framework no longer limited to network characterisation. More precisely, we have made extensive changes to this library (>90% of the code base) to cater for most of the previously mentioned requirements (namely, Req. 1 to 6). We believe that the resulting OML2[1] software is a good candidate for an SDM framework.

The objective of this article is manifold. First we propose a generic architecture towards which we believe an SDM framework should tend. We also provide a comprehensive overview of the OML design towards implementing this SDM architecture. Second, we offer an experimental characterisation of the bias that OML might introduce to systems under study, thus evaluating its compliance to Req. 1. This allows us to derive a set of guidelines to ensure that they do not reach a statistically significant level.

The remainder of this article is organised as follows. Section 2 presents work related to measurement and collection frameworks for experimental studies. In Section 3, we present our proposed conceptual framework, as well as its implementation into OML. We also review the API of this suite, and highlight the steps needed to create an instrumented application. We then evaluate OML's "observer effect" in Section 4 and derive usage guidelines to minimise or remove it when present. Finally, Section 5 concludes this article and highlights future directions towards a full implementation of the proposed framework.

2 Related Work

A distributed monitoring framework usually comprises three generic elements: probes performing the actual measurement data collection, formats and protocols allowing for storage and exchange of the measurements, and tools to process, forward and store the samples. In this section, we review the state of art for these different elements. A summary of this discussion in regards to each tool's compliance with SDM requirements defined in Section 1 can be found in Table 1.

2.1 Data Collection Tools

The networking community has been developing and using several types of stand-alone measurement tools dedicated to a specific task, such as `tcpdump`(1), D-ITG [4, 5] or

[1] In the remainder of this article, we simply use "OML" to refer to this suite, available at http://oml.mytestbed.net.

Table 1: Compliance of existing systems with the requirements for an SDM framework. '–' indicates irrelevance, and '?' unclear or lack of information.

System	Req. 1	Req. 2	Req. 3	Req. 4	Req. 5	Req. 6	Req. 7	Req. 8
Data collection tools								
Iperf [17]	some (Sec. 4)	no	some	–	no	no	no	no
D-ITG [5]	no [4]	no	no	–	no	no	no	no
`tcpdump`(1)	?	local	no	–	no (PCAP)	no	no	no
DAG	?	local	?	–	no	no	no	no
DTrace [11]	no	local	some	yes	some	some	no	no
Zabbix [33], Zenoss [6]	?	local	some	yes	some	some	some	some
Data format and protocols								
PCAP	–	local	some	–	no	no	no	–
SNMP [19]	–	no	no	some	no	some	some	–
IPFIX [14]	–	local	no	some	yes [9]	some	some	–
Processing and reporting frameworks								
DIMES [47]	?	?	?	no	no	yes	no	no
PlanetFlow [20]	low	local	some	no	no	yes	no	no
CoMon [35]	?	local	no	?	no	yes	yes	no
CoMo [22]	?	local	no	no	no	some	some	no
MINER [10]	?	?	yes	some	some	some	some	
DipZoom [38]	low [50]	?	?	some	some	yes	yes	no
PerfSONAR [18]	?	?	no	yes	some	yes	yes	some
BlockMon [21]	?	?	no	no	some	no	no	no
OML	low (Sec. 4)	some (Sec. 3)	yes	yes	yes	some	some	no

Iperf [17]. The former has been shown to accurately report at capture rates up to gigabits per second [44] while the latter allows researchers to generate a traffic load to evaluate the capacity of a network or the resilience of a system. In particular the authors of [25] showed that Iperf generated the highest load on network paths compared to other traffic generators. High performance or versatile hardware solutions have also been developed, such as DAG,[2] but usually store the data locally and thus do not comply with Req. 6.

As we noted in the introduction, one common problem with these tools is that they do not share output formats, and post-processing is required before being able to cross-analyse their data, thus hindering Req. 4 and 7. The problem of data collection from distributed nodes (Req. 6) is also not addressed at this level. Additionally, there is little or no study characterising potential biases or impact of using these tools on the system under study (Req. 1). In particular, we show in Section 4 that under certain conditions, using OML to report Iperf's results rather than its standard CSV output significantly increases the achievable throughput.

Several solutions exist to support more generic data collection (Req. 5), allowing for instrumentation and metrics collection from various networking applications and devices. A few system monitoring tools such as Zabbix [33] or Zenoss [6] can rely on the measurement of various tools (Req. 4) to derive the health of a system, report it remotely (Req. 6) and/or take some corrective actions (Req. 8). Two more generic data-collection tools are SNMP [19] and DTrace [11]. Similar to our proposal, they both allow the instrumentation of any software and/or devices. In addition, DTrace can dynamically instrument live applications, and is shipped by default with some operating systems. However, its measurement processing (Req. 6) is limited to aggregating functions, and it does not support the streaming of measurements from different devices to a remote collection point (Req. 7).

2.2 Data Format and Protocols

Though there is little agreement about how measurement data should be stored, a notable commonality is `tcpdump`'s PCAP format. This format is often limited to the timestamped representation of packet traces (failing to cater for Req. 2), but many tools are able to open, process and write files in this format (*e.g.*, Wireshark,[3] or `libtrace` [2]).

SNMP has been widely adopted for the management and monitoring of devices, and allows the collection of information over the network. However, it has some performance and scaling limitations when measurements from large number of devices are required within a short time window [54], which does not comply with Req. 1. SNMP is also constrained to only reporting information predefined in its management information base (MIB), which limits its support for Req. 5.

IPFIX [14] is an IETF standard, replacing Cisco's NetFlow [13], which leverages SNMP's MIB and defines a protocol for streaming information about IP traffic over the network. Similar to OML clients, IPFIX exporters stream collected and potentially filtered measurements to collector points (Req. 6). While IPFIX was initially limited to measurements about IP flows, an extension [9] allows custom types to be specified for the reported data, allowing compliance with Req. 5. This extension however relies on external information describing the semantics of the newly-defined information elements. Other serialisation protocols such as XDR [15] or ASN.1 [1] support similar functionalities.

2.3 Processing and Forwarding Frameworks

More complete measurement frameworks have also been proposed for distributed observations. In particular, DIMES [47] allows measurement probes to be deployed throughout the Internet, but is limited to topology information which goes against Req. 5. PlanetFlow [20] and CoMon [35] provide flow logging and slice or node monitoring for PlanetLab (limited match with Req. 5), including sophisticated query mechanisms (Req. 6).

CoMo (*Co*ntinuous *Mo*nitoring) [22] is a network measurement system based on packet flows (in contradiction with Req. 5). It has core processes that are linked in stages, namely *packet capture*, *export*, *storage*, and *query*. These processes are linked by user-defined modules, used to customise the measurement system and implement filtering functions on the captured packet traces (satisfying parts of Req. 6). OML, by contrast, is a generic framework that can instrument the whole software stack, and take input from any sensor with a software interface.

DipZoom [38, 50] allows third-party measurement providers to be leveraged in order to perform measurement studies at the scale of the Internet. Using the analogy of a market, it establishes links between users offering the desired measurement tool (Req. 4 and 5), from a given vantage point, and the experimenter trying to evaluate a hypothesis (Req. 6 and 7).

PerfSONAR [18] proposes to extend system and network monitoring by allowing multiple administrative domains to share and access each other's measurements (Req. 6 and 7) collected through various legacy tools (Req. 4) or protocols (*e.g.*, NetFlow or SNMP).

BlockMon [21] is a composable passive network analysis tool. While its GUI makes it easy to use, its sole focus on network traffic (against Req. 5) and local collection and processing (against Req. 6) do not make it a good SDM candidate.

Of all the measurement schemes that we surveyed, MINER [10] appears to be the closest to the framework proposed in this paper. It comprises a measurement architecture as well as elements of a management framework. The MINER tools are Java components that may provide measurement results directly, or may be wrappers around external libraries or applications that do the actual measurements (Req. 4). Unfortunately, MINER is not open source software, which limits its extensibility to other types of measurements (Req. 5).

Next, we propose a conceptual framework allowing to support all SDM requirements, and present the state and use of its implementation within OML.

[2] http://www.endace.com
[3] https://www.wireshark.org

3 Architecture

At the core of a distributed measurement architecture is a stream of measurements from the element observing a metric to destinations interested in this metric, either for processing, storage, display or other use. This section describes such measurement streams, then presents basic building blocks of an architecture to exchange these. An implementation of a subset of these concepts into a protocol and its associated instrumentation library is also presented here.

3.1 Measurement Streams

A single measurement from a sensor yields one or more metrics. A repeated measurement over time provides a sequence of such samples. Rather than considering singular metrics, it is therefore desirable to be able to deal with groups of samples—tuples—of one or more variables, which we thereafter call *measurements*. A sequence in time of these measurements, sent over the network, is therefore a *measurement stream* (MS).

An MS is therefore defined as a sequence of tuples with a constant *schema*. A schema comprises at least names and types (integer, floating point, string, blob, etc.). Each measurement is also timestamped with the time of its sampling or, if missing, that of its introduction into the stream. Schemata naturally map to database columns, while each measurement can then be placed as a row in the thus-defined table. This combination of data tuples with a regular format and a timestamp makes these MSs rather similar to the concept of streaming databases [3, 12].

Schemata can also be enhanced with information about units and precisions, which are particularly important for proper curation and reusability of the collected data. Additionally, more generic information, changing at a slower pace, can further describe the context of the measurement (*e.g.*, location, software version or serial number) and help qualify the measurements. Additional MSs can be used to convey such metadata.

3.2 Measurement Framework

Based on the definition of a measurement point, the measurement framework provides two basic elements, *sources* and *sinks*, which respectively generate and consume MSs. On the wire, MSs can be encoded by different protocols, provided they are expressive enough to convey the properties presented above.

The sources and sinks only provide an abstraction for the interaction with measurement streams, but do not perform any active task on them. For this purpose, four generic data manipulation points are envisioned.

Injection points are sources; they consist of any kind of tool for or through which some metric can be observed, instrumented in order to generate one or more MSs;

Processing points are sinks receiving one or more MSs, processing them in some definable way (*e.g.*, aggregation or statistical analysis); they thus generate new MSs for which they act as a source;

Collection points are terminating sinks; they include sinks and some functionality, out of the scope of the reporting framework, which do not generate more MSs; typical examples include storage endpoints or visualisation dashboards;

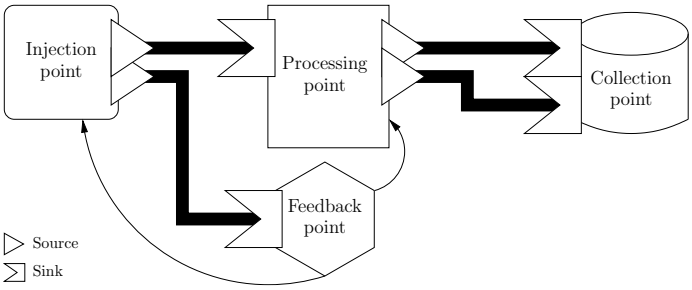

Figure 1: Generic manipulation elements of the proposed measurement framework. An injection point creates two MSs (thick lines). One of them is handled by the processing point which creates two streams out of it, and sends them to the collection point. The other stream from the injection point is sent to a feedback point which controls (thin arrows) some aspect of the measurement tool at the injection point, as well as some of the processing point's parameters.

Feedback points are a specialisation of collection points; they perform analytic tasks on the received MSs and provide feedback to the measurement infrastructure itself; they primary objective is to support adaptive measurement regimes or steerable experiments.

Figure 1 illustrates a setup using each of these four elements. These elements are distributed among the nodes either complementing measurement tools or as stand-alone application. In the particular case of OML, the injection points are usually on the sensor side with some processing points, while the collection and feedback points are implemented as separated entities either on the same node or a distant server. This organisation in OML is illustrated in more detail in the following section. Furthermore, these conceptual points do not require any support from the network.

Next, we present the OML suite, which provides functionalities to instrument application and enrol them into the presented architecture. While it was originally an integral part of OMF [40, 41], which it was developed to support measurement for, it is a stand-alone instrumentation suite which can be used in other contexts. It is currently used within GENI [7, 8] and various EU projects such as Fed4FIRE [30], OpenLab [32] or SmartSantander [43], amongst others.

3.3 Implementation

The version of OML at the time of this writing (2.11) only implements a subset of the proposed framework. Following the structure of the presented architecture, OML defines a protocol which allows measurement streams to be transported on the network. Though the protocol could be implemented directly by an application, OML provides a set of mature tools and libraries to simplify this task.

```
protocol: 4
experiment-id: localdomain
start-time: 1340855573
sender-id: localhost
app-name: iperf
schema: 0 _experiment_metadata subject:string ↩
    key:string value:string
[...]
schema: 4 iperf_transfer pid:int32 connection_id:int32 ↩
    begin_interval:double end_interval:double size:uint32
schema: 5 iperf_connection pid:int32 ↩
    connection_id:int32 local_address:string ↩
    local_port:int32 remote_address:string ↩
    remote_port:int32
schema: 6 iperf_settings pid:int32 server_mode:int32 ↩
    bind_address:string multicast:int32 ↩
    multicast_ttl:int32 transport_protocol:int32 ↩
    window_size:int32 buffer_size:int32
schema: 7 iperf_application pid:int32 version:string ↩
    cmdline:string starttime_s:int32 starttime_us:int32
content: text

0.600149⟶7⟶1⟶32438⟶2.0.5+oml1⟶iperf -c ↩
    localhost⟶1340855573⟶600144
0.601109⟶6⟶1⟶32438⟶1⟶0.0.0.0 0⟶0⟶6⟶↩
    172760⟶131072
0.601139⟶5⟶1⟶32438⟶4⟶127.0.0.1⟶39757⟶↩
    127.0.0.1⟶5001
10.610647⟶4⟶11⟶32438⟶4⟶0.000000⟶↩
    10.000086⟶74317824
```

Listing 1: Example MSs generated by Iperf using the OML Measurement Stream Protocol. The header identifies the protocol version and the source, as well as the schemata of the streams; not all the schemata defined are used in this example. The first column is a timestamp (relative to the **start-time**) of the introduction of the tuple in the stream, the second identifies the schema of the row, and the third is a sequence number. The remainder of each line corresponds to the columns of the specified schema. After some initial tuples containing extra information about the run (schemata 5–7), a first throughput sample is streamed (schema 4).

3.3.1 OML Measurement Stream Protocol

The OML Measurement Stream Protocol (OMSP) consists of a small header followed by series of encoded MS tuples (serialised either as plain text or using a binary encoding). The current protocol is designed for one-way lossless point-to-point connections—in other words, TCP. However, its design also lends itself to be used over UDP-based multicast.

Listing 1 shows an example of the text protocol. The straightforwardness of this encoding, made of tab-separated values, makes it easy to write an application generating or consuming MSs. The binary protocol, specified in [51, ch. 9],[4] is available for more data intensive use, as it tends to be more compact on the wire. The latter form of the protocol is generally the default for the software instrumentation tools provided by OML and presented next.

3.3.2 OML Instrumentation Tools

OML is a distributed and multi-threaded instrumentation suite allowing MSs to be represented and manipulated in the protocol just presented. Several elements are provided, which map to the basic manipulation points of the presented framework, as illustrated in Figure 2. OML is available as open source under an MIT license, making it easier to adapt in case specific needs require it [*e.g.*, 26].

[4]The latest OMSP specification is also available at http://oml.mytestbed.net/doc/oml/latest/doxygen/omsp.html

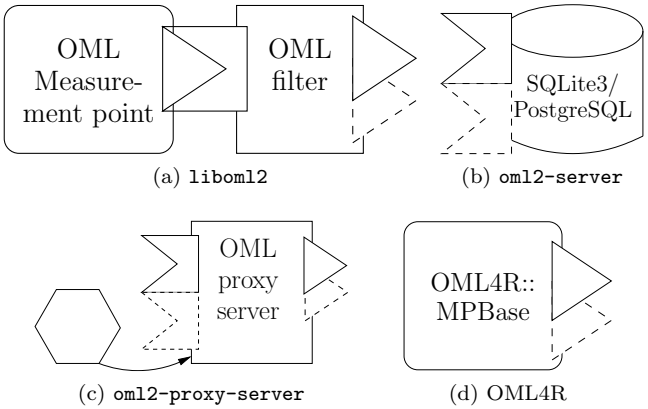

Figure 2: Functionalities of the OML instrumentation library and tools as components of the measurement framework.

`liboml2(1,3)` is a C library providing an API to instrument any tool, making them injection points; in addition, it implements some filtering capabilities akin to those of a processing point (Figure 2a).

`oml2-server(1)` is a collection point which can store measurements streams into databases such as SQLite3 or PostgreSQL, as well as iRODS [39] (Figure 2b).

`oml2-proxy-server(1)` allows the path between injection and collection points to be decoupled. It also provides buffering capabilities for disconnected measurement cases [52]; as such it is an instance of a processing point with some feedback control based on the connectivity state (Figure 2c).

OML4R is a native Ruby implementation packaged as a Gem[5] to create injection points (Figure 2d).

OML4Py is a Python library similar to OML4R available on PyPi.[6]

OML4x**,** where $x \in$ J, Node, JS, are other similar injection point libraries (Java, Node.js, JavaScript) currently being developed by third parties.

`liboml2(1,3)` and `oml2-server(1)` The client library can be used to instrument any piece of software. An instrumented application injects its samples to the library for processing and streaming to at least one collection server. When requested by the experimenter, this client library may apply some processing filters to the measurements before forwarding them (Figure 2a). The collection server receives the samples from software running on all nodes involved in a given experiment (identified by a unique ID), and can store them, in a unified timestamped format, in various data-storage backends such as SQLite3 or iRODS (Figure 2b).

Within an experiment, each OML-instrumented tool adds an injection timestamp (`oml_ts_client`) to generated samples, with reference to the nodes' clock. The server adds

[5]https://rubygems.org/gems/oml4r
[6]https://pypi.python.org/pypi/oml4py

a reception timestamp (`oml_ts_server`) based on its local clock reading. Both timestamps are stored in the database rebased to the `start-time` of the experiment, *i.e.*, the time of the first connected client for that experiment. Comparisons between `oml_ts_client` from the same node are exact. If samples from different nodes are considered, the accuracy of the comparison depends on the precision of the time-synchronisation protocol (*e.g.*, $\sim$ 10 ms with NTP accross the public Internet [16]); a higher-resolution comparison can also be achieved by deploying a separate synchronisation scheme such as [49] or GPS hardware on all clients. Comparisons between `oml_ts_server` are exact, but should take into account the network delay from each node to reach the server.[7]

To instrument an application with OML, a developer first defines measurement points (MPs) within its source code (Listing 2 shows an example of such instrumentation). An MP is an abstraction for a tuple of related metrics to be reported at the same instant. Thus the MPs define all the potential MSs that the application can generate. The modified source code is then compiled against the OML client library to generate the OML-instrumented software. At run-time, the experimenter can activate some or all of the defined MPs to generate MSs. This is done through an XML configuration file passed to the software at start-up (as documented in [51, sec. 3.1.2]). Any application instrumented with `liboml2`(1) also sees its command line arguments and environment variables extended by OML-specific options to configure parameters such as identifiers and collection servers.

Prior to streaming MSs to a the `oml2-server` or other collection point, the client library may apply predefined filters to the samples, as described in the XML configuration file. This filtering process is illustrated in Figure 3. Filters are composable functions which are applied to a subset of metrics from MSs over a given time or sample period (*e.g.*, every 1 s or 10 samples). Thus they integrate incoming MSs into newly generated outgoing MSs, as processing points do (hence Figure 2a, where `liboml2` is represented as both an injection and a processing points). Some built-in filters are available by default, and can be complemented by custom users-written ones. Examples of simple OML filtering capabilities include averaging a metric over a time window, or getting its extreme values (min/max) over a given number of samples.

The implementation of OMSP over TCP has finite-sized outgoing buffers and may therefore drop some measurement samples if the path to the server cannot provide sufficient capacity to cater for the sample rate. A sequence number incremented for every sample allows the detection of such events on the server side. The `oml2-proxy-server`, presented below, can however alleviate this issue.

We evaluate the performance of this implementation, and its impact on the behaviour of instrumented applications in Section 4, while the following text presents the two remaining elements of the OML suite, namely the `oml2-proxy-server` and the OML4R Ruby Gem and other native implementations.

[7] This paragraph has been amended on 2014-03-12 to correct inaccuracies in the original article.

```c
/* 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->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 */
}
```

Listing 2: Using `liboml2` to instrument an application can be done with only a few lines of code. Helper files and functions, such as `example_oml.h` and all `oml_*()`, are generated by `oml2-scaffold`(1) from a Ruby-like description of the applications' measurement points (see Listing 4 in A). This code can be compiled with `$CC example.c -o example -loml2`.

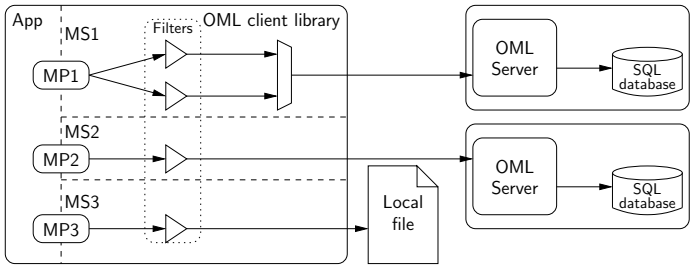

Figure 3: Measurement data path in OML: Three MPs are filtered to generate MSs which get sent for storage at different locations.

`oml2-proxy-server`(1) One of the primary aims of the filtering facility of `liboml2` is to allow the reduction of the measurement data that needs to be transmitted to the server. However, in some circumstances the experimenter might want to observe effects that the filtering would discard. In that case another solution is required.

Even in networks with a static topology, it can become necessary to buffer measurement data. If the experiment involves high traffic rates on high speed interfaces, the rate of generation of measurements may also be very large, and the data path to the measurement server can become congested, risking loss of measurement data.

Recalling that this measurement framework has to be as convenient as possible for researchers to use, it is important to ensure that buffering measurements does not force complicated modifications to the client applications. Our solution is to create a separate proxy server as a controllable processing point (Figure 2c) acting as a FIFO queue

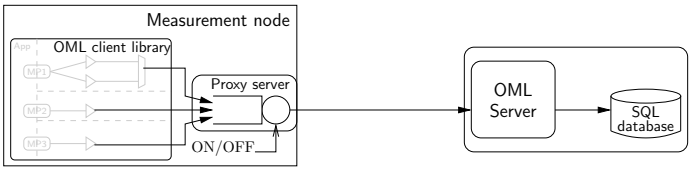

Figure 4: Measurement architecture where the path from injection to collection is decoupled by the use of an `oml2-proxy-server`, *e.g.*, for experiments with mobile nodes experiencing periods of lack of connectivity.

which network output can be temporarily disabled, and the data temporarily buffered locally [52].

The `oml2-proxy-server` presents an interface to the client applications that is identical to the `oml2-server`. It is thus transparent to the client applications, which do not need to be modified or re-compiled. In Figure 4, the proxy server is shown running on the same node as the experiment applications, but it can be hosted on a separate node if the situation requires. At the end of the experiment, the experimenter instructs the proxy server to turn the output stream "on", whereupon the proxy server connects to the upstream collection point and transmits the buffered streams to it. Data-based feedback is however a feature missing from the `oml2-proxy-server` as of versions 2.6–2.11 of OML.

The proxy server implementation is simple, as it does not have to process any data on its input stream. Furthermore, since it is a one-way stream, the proxy server can simply store raw octets from the experiment applications in memory, and then replay them out to the OML server verbatim. It also has the option to write the measurement data to disk to provide a backup and limit its memory usage.

The `oml2-proxy-server`(1) can be deployed on any machine, its command line argument indicating both where it should listen for OML streams, and where it should forward them. `PAUSE` and `RESUME` instructions can simply be sent to its standard input. More details can be found in [52].

Other Native Implementations OML injection points libraries have also been implemented natively for other languages. The most complete are OML4R (as shown in Figure 2d) and OML4Py for Ruby and Python applications, respectively. They provide a base class for an OML measurement point (`OML4R::MPBase` or `oml4py.OMLBase`) from which the developer or instrumenter can inherit to create classes for specific MPs. The rest of its usage is essentially similar to that of `liboml2`(3), as illustrated in Listing 3 for OML4R. However, contrary to `liboml2`, these libraries only support the text protocol, and do not provide any filtering capability at this stage. Other native libraries are currently being developed for Java, JavaScript or Node.js.

The OML-related command line options and environment variables of the thus-instrumented Ruby or Python tools are similar to those of `liboml2`(1), therefore providing a consistent interface to all instrumented applications.

The character string-parsing capabilities of these scripting languages, combined with the relevant native implementation, particularly allow the experimenter to write wrapper scripts. Such scripts can read and process the output from any applications—without limitation on the availability or modifiability of their code—extract the relevant metrics, and report them through OML. An example of such a usage with OML4R, a wrapper around `ping`(8), can be found in B.

```
# oml4r-example.rb
require 'rubygems'
require 'oml4r'

class ExampleMP < OML4R::MPBase # Register the MP
  name :example_mp
  param :label                  # :type is :string by
      default
  param :n, :type => :int32
end

opts = {:appName => 'example'}  # Additional,
    optional, parameters include :expID, :nodeID and
    :omlServer
OML4R::init(ARGV, opts)         # Initialise OML,
    establish a connection, and send headers

while true
  label, n = application_measure_something
  ExampleMP::inject label, n    # Inject tuple
  do_some_other_things
end

OML4R::close                    # Join the threads
    and empty send buffers
```

Listing 3: Using OML4R in Ruby to the same effect as Listing 2 in C.

4 Performance Evaluation

This section presents our evaluation of two performance aspects of OML (version 2.6.1). First, we assess the potential observer effect induced by `liboml2` on the software being instrumented. Then, we quantify the impact of its sampling rate at scale when the testbed does not provide a separate control network to carry OML measurement streams.

The experimental scenarios presented below are not novel. However, their purpose here is purely to load the reporting chain and allow us to to evaluate whether it has undesirable effects on the systems under study. We note that, while all these experiments involve rather basic observation of network flows, OML is not limited to these use cases.

4.1 Quantifying the Observer Effect Induced by OML Instrumentation

One of the main issues in designing an experimental study is to ensure that collecting measurements will only insignificantly disturb or modify the system under study. Thus, we performed an extensive analysis of the impact of OML on the performance and behaviour of instrumented applications [27]. The main findings from that work are the operative ranges and configuration where OML does not significantly ($p < 0.05$) impact the systems under study nor biases the results. They are summarised below.

4.1.1 Method

While researchers have used OML to instrument many software tools,[8] we limit this study to two of them, namely the

[8] See, *e.g.*, http://omlapp.mytestbed.net or http://witestlab.poly.edu/index.php/wirelesstestbeds/omfnettest/oml-applications.html.

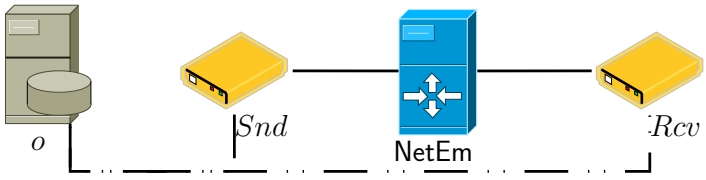

Figure 5: Simple Gigabit two-hop topology used for the performance evaluation of the impact of OML reporting on instrumented applications.

Iperf network probing application [17], and the `libtrace` packet capture library [2]. We emphasise that the purpose of this study is not to evaluate these tools, but rather to assess if using OML to report their generated measurements introduces any deviations in their performance and response.

In that regard, we designed two sets of experiments based on the simple two-hop topology shown in Figure 5. This topology is composed of a sender and a receiver exchanging Iperf-generated traffic via a router, over Gigabit interfaces. A non-instrumented `tcpdump` observes the traffic on each interface of the router. A separate control network is used to carry generated measurements to an `oml2-server`.

Iperf Instrumentation Our first experiment set focuses on OML's effect on the Iperf probing tool. The original Iperf can report detailed network measurements in a CSV file. We modified this reporting procedure to make Iperf use `liboml2` to report its measurements instead [27].[9] While many factors may influence the performance and behaviour of Iperf, in this study we limit our experimental design to three factors. The first one is the **flavour of Iperf** being used, which can be one of the following.

nooml the original Iperf,

o OML-enabled Iperf, reporting metrics computed by Iperf,

O OML-enabled Iperf, reporting all sent/received packets, and

Of OML-enabled Iperf, computing aggregates over all sent/received packets with OML filters.

Another factor is the **requested sending rate** at which Iperf is instructed to send packets. Although this rate is a continuous variable, we are only interested in given rate values in this study. Thus we treat it as a fixed factor with the values: 10, 50, 75, 100, 200 and 300 Mbps.

Our last factor for this experiment set is the use of threads or not in Iperf's reporting process.[10] The values for this factor are threads and nothreads.

We measure three dependent variables in this experiment set. The first one is the **actual sending rate** (R_t^{RtrIn}) of the sending Iperf as computed from `tcpdump` traces (t) on the router's ingress link from the sender. The second measured variable is the **accuracy of the throughput report** (T_{Diff}^{Rcv}) of the receiving Iperf (i), which is the difference between the throughput as reported by the receiving Iperf (T_i^{Rcv}) and the throughput computed from `tcpdump` traces on the router's egress link to the receiver (R_t^{RtrEg}).

For a given sample,

$$T_{\text{Diff}}^{Rcv} = R_t^{RtrEg} - T_i^{Rcv}. \quad (1)$$

The last variable considered is the **jitter report accuracy** (J_{Diff}), which is the difference between the jitter as reported by the receiving Iperf (J_i^{Rcv}) and the jitter computed using the jitter equation from [45] (which is the same as what Iperf uses internally [27]) on the `tcpdump` traces on the router's egress link to the receiver (J_t^{RtrEg}). For a given sample, we have

$$J_{\text{Diff}} = J_i^{Rcv} - J_t^{RtrEg}. \quad (2)$$

`libtrace` Instrumentation In a second set of experiments, we focus on OML's impact on a `libtrace`-based packet capture application (`trace-oml2`). More particularly, we are interested in the potential degradation on the accuracy in packet reporting from the `libtrace` instrumentation, as well as their timestamps. As in the previous experiment set, the first factor that may influence these variables is whether the instrumentation is enabled or not, which can be one of the following, `trace-nooml` (nooml; using CSV reporting), `trace-oml2` (oml) and an extension to the latter which uses of a summation filter (Of) on the length field of the IP header (`ip_len`) to compute the traffic rate. Similarly, the second factor in this set is the requested sending rate for the generated traffic between the sender and the receiver, with the same possible values as in the previous experiment set.

We also measure three dependent variables in this experiment set. The first one is the **accuracy of packet reports**, in the form of losses (L^{Rcv}). This is the ratio between the number of packets sent to the receiver as counted from `tcpdump` traces on the router's egress link (N_t^{RtrEg}) to that of the packets reported by the receiver's instrumentation (N_i^{Rcv}). Thus for a given sampling window, we have

$$L^{Rcv} = 1 - \frac{N_i^{Rcv}}{N_t^{RtrEg}}. \quad (3)$$

Closely related, our second measured variable is the **accuracy of received rate** (R^{Rcv}), which is computed using the `ip_len` field of the reported packets,

$$R^{Rcv} = \sum_0^N \texttt{ip\_len}. \quad (4)$$

Our last measured variable is the **accuracy of timestamp reports** (t_{Diff}^{Rcv}). For a given packet this is the difference between the timestamp from the `tcpdump` traces and the one from the `libtrace` report at the receiver,

$$t_{\text{Diff}}^{Rcv} = t_t^{Rcv} - t_i^{Rcv}. \quad (5)$$

4.1.2 Results

All experiments were performed on an OMF testbed [40, 41] controlled with the IREEL experimentation portal [23]. Their precise descriptions are available[11] and can be used

[9] http://omlapp.mytestbed.net/projects/iperf/wiki

[10] This is an option already present in the original Iperf.

[11] http://ireel.npc.nicta.com.au/projects/omlperf/wiki

to reproduce the study on any OMF-enabled testbed. Recent machines[12] with two separate Intel Pro/1000 Gigabit LAN interfaces carried the experimental traffic on one side and the control and measurement traffic on the other. This ensured that neither the network traffic nor the hardware capability of the machines[13] biased our experiments. UDP was used to as the experimental traffic in order to allow jitter analyses.

This section presents the results of the analyses which we performed on the data collected from these experiments. All of these collected measurements are available online, along with the R [37] scripts used to analyse them.[11] In each 5 min run, we collected 600 aggregate samples (one every half-second) for each parameter combination in each experiment set. The sample times where rebased to 0 in their respective time frame to allow for meaningful comparisons between sources. The first and last samples of each data set were ignored to avoid bias due to incomplete measurement periods at start up and tear-down.

As our experiments involved factors with categorical values and continuous dependent variables, we used analyses of variance (ANOVA) to compare the influence of each factors. Some of the ANOVA assumptions were not met by some parts of our data; details on how we addressed these issues can be found in [27].

Iperf Instrumentation

We performed two-way ANOVAs with interactions for each of the variables R_t^{RtrIn}, T_{Diff}^{Rcv} and J_{Diff} at each of the studied set rates. Due to space constraints, we only focus here on the statistically significant ANOVA results. The complete set of analysis results is available in [27].

Actual Sending Rate For rates 75 Mbps and higher, there are statistically significant differences ($p \leq 0.05$) in Iperf's sending rates, R_t^{RtrIn}, which are introduced by changes in both the use of threads and OML instrumentation, as well as the interaction of those two factors. As this interaction is significant, we study it first, in Figure 6 for case 300 Mbps. This figure shows the difference in means of the response variable. It shows an interaction between the threads and oml factors. The O factor introduces a sizeable negative impact, which is only partially mitigated by Iperf's native threads. However, the use of filters in the Of factor completely removes the issue.

The Tukey Honest Significant Differences test confirms that using OML to report every packets (O) has a significant impact on the sending rate, reducing it by an average (at most) 10.3 MBps at set rate 300 Mbps (about 27.5%). Iperf's internal threads can reduce this difference by 4.17 MBps, which is still a 16.3% drop from the performance for the other treatments. In all cases, use of OML in-line filtering (Of) completely removes the problem. Interestingly, though it is not found significant here, the use of OML filters in the Of treatments consistently enabled a slight increase in the sender's rate as compared to vanilla Iperf (nooml treatment).

[12] 3.20 GHz Pentium 4 processors, with 2 GB of RAM running Ubuntu Linux with kernel `2.6.35-30-generic #59-Ubuntu`

[13] The PCI buses on our experimental machines were limited to 500 Mbps full-duplex, which was taken into account in our choice of treatments for the sending rate factor.

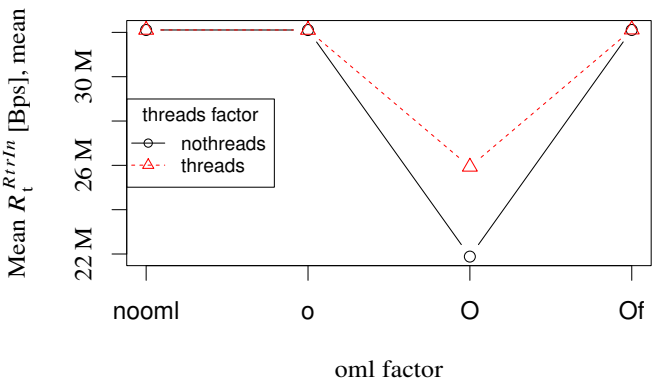

Figure 6: Graph of means for the interaction between experimental factors on R_t^{RtrIn} at 300 Mbps. While advanced OML appears to have a negative impact, it is partially mitigated by Iperf's internal threading, and fully removed by the use of OML filters

Accuracy of Throughput Reports Here, we assess the variations of T_{Diff}^{Rcv} (1) between the treatment groups, as an evaluation of the impact of the instrumentation on the accuracy of Iperf's report.

At rates 10–200 Mbps, no statistically significant ($p > 0.05$) difference can be found. Only for set rate 300 Mbps do important ($p \leq 0.05$) deviations in the mean appear. The combination of the OML and threads factors is studied first. The graph of means is qualitatively similar to Figure 6. Tukey HSD tests confirm that the treatment causing this deviation is also the non-filtered advanced mode (O) in both threads and nothreads treatments. No other difference in mean between other treatments (particularly nooml and Of) is found to be significant ($p > 0.05$).

Accuracy of Jitter Reports We finally attempt to find differences in J_{Diff} (2) in a similar fashion. As OML does not currently provide a jitter-computing filter, we could not consider treatment Of in this case. For treatment O, we post-processed the packet records based on their arrival times to compute the jitter equation use by Iperf and described in [45].

For rates 10 and 50 Mbps, no significant difference in the means could be found ($p > 0.05$). For rates 75 Mbps and higher, however, the analyses of variance identified statistically significant ($p \leq 0.05$) deviations. As before, they were always linked to the Iperf advanced mode (O), in comparison to the vanilla and legacy report modes. This difference can be explained in a similar fashion as for the throughput reports where, with an increasing number of packets not being reported, the computed metric loses accuracy. A jitter-computing filter for OML would address this issue in the same way as the sum filter did in the previous section.

`libtrace` Instrumentation

We performed a similar analysis on the data from our second experiment set. Statistically significant ($p < 0.05$) results are reported here.

Accuracy of Packet Reports In this experiment, we are first interested in the loss ratio L^{Rcv} (3). In all

Table 2: Packet timestamp difference between PCAP capture via `libtrace` and OML report timestamp (`oml_ts_client`) generated by `trace-oml2`.

Set rate	t^{Rcv}_{Diff} [s]				
[Mbps]	min.	med.	avg.	max.	sd
10	-7.0 n	0.072 μ	0.10 μ	0.47 μ	0.10 μ
50	-5.5 n	0.069 μ	0.098 μ	0.47 μ	0.099 μ
75	-5.7 n	0.077 μ	0.11 μ	0.47 μ	0.11 μ
100	-7.0 n	0.071 μ	0.10 μ	0.47 μ	0.10 μ
200	0.0	0.0	3.7	132.9	21.9
300	0.0	0.0	6.3	162.8	31.4

treatments of the rate factor (10–300 Mbps), the OML-instrumented packet-capture application's reports deviate in a statistically significant manner ($p \leq 0.001$) from the non-instrumented version. This application reports two samples (`ip` and `udp` MPs) per captured packet. With packets of size 1,498 B, this induces a rate between 834 and 25,034 pps, and double the number of samples. On average, 7.75 pps went unreported at 10 Mbps (0.93 %), but this went up to 216 pps at 300 Mbps (only 0.86 %).

As no loss filter is currently available for OML, only the `nooml` and `oml` treatments were studied for the OML factor. Considering the reported rate R^{Rcv} (4), as computed by summing the IP length of the reported factors, allows us to provides some insight nonetheless. As for the losses, the throughput exhibited significant differences ($p \leq 0.001$) depending on whether it was computed from `trace-nooml` or `trace-oml2`'s reports, with the latter being consistently lower (between 0.08 and 2.21 %). However, complementing the use of `trace-oml2` with a summing filter (`Of`) produces statistically significant ($p \leq 0.001$ for treatments 75–300 Mbps) *positive* differences. We hypothesise that limiting processing in the main thread and reducing the number of report packets to be sent allowed for more packets to be read on time from the packet capture buffers, resulting in an increase in the reported throughput by 0.15–0.85 % depending on the cases. These observations are consistent with the Iperf results from the previous section.

Timestamp Accuracy and Precision The `trace-nooml` tool does not compute a local timestamp as OML does for `trace-oml2`. It is therefore not possible to obtain t^{Rcv}_{Diff} (5) for the `nooml` case for comparison. Rather, we only consider the `oml` treatment. and give summary statistics for t^{Rcv}_{Diff}. They are summarised in Table 2. For rate 100 Mbps and below, the time difference is almost negligible, with a maximum at 0.47 μs, and a mean of about 100 ns. It is interesting to note that some minimal differences are *negative*, which would hint that slightly different clocks are involved in the kernel-land timestamping of PCAP packets and OML's user-land `gettimeofday`(3) requests.

The picture is clearly different for rates 200 and 300 Mbps. For these treatments, the maximum difference is more than 2 minutes and the average time differences are of the order of several seconds. We recall that at these rates, 30–50 k samples are generated per second. These samples are stored, on the server side, in a FIFO queue, before being entered in a database. We hypothesise that this is an indication that `oml2-server`, as shipped with version 2.6.1 of the suite, cannot handle such high loads and effectively breaks before that. If per-packet precision is not required, aggregation filters can however be used to lift this limitation.

4.1.3 Discussion

In this study the underlying two null hypotheses were as follows: "the OML instrumentation of Iperf has no significant effect on its packet-sending rate nor on the accuracy of its throughput and jitter reports" and "the OML instrumentation of `libtrace` has no significant effect on the accuracy of its packet and timestamp reports".

In the case of the Iperf's instrumentation, results show that the OML-instrumentation of the legacy reporting mode of Iperf (treatment `o`) does not introduce any significant deviation from the normal behaviour, at any rate. However the advanced per packet reporting mode (`O`) is more intrusive and does introduce a negative bias in the reported metrics and the general behaviour of the application when used without care. The introduction in OML's reporting loop of aggregating functions such as a sum filter (`Of`) completely alleviates the issue.

For the `libtrace` instrumentation, results show a similar trend, where per-packets reports at very high rates have significant differences from what would be measured without OML. Once again, the proper use of upstream processing filters can cancel out this problem. Moreover in this case, we found a statistically significant positive bias introduced by the use of filters, as it allowed the packet-capturing tool to resume reading the capture buffer faster, while OML was processing the samples in a separate thread.

Whilst the experiments presented in this study focused on the OML effect on the instrumented measurement tools, it would be legitimate to also question the effect of this library on the physical resources themselves. In order to evaluate this effect, we performed a pilot study and found no significant impact on neither the CPU nor the memory usage when both applications were reporting with OML [29]. For the sake of space and clarity, we opted not to include these results in this article.

It is worth noting that, on the less powerful machines we used in that first study, we also found a positive significant impact of OML when used with thread-less versions of Iperf. In these cases, using OML instead of Iperf's normal report channels would bring the performance of the non-threaded flavours to that of the threaded ones.

Based on these results, some general guidelines can be derived. There is a clear causal link between the increase of the sample rate, and the decrease in performance of an instrumentation. An instrumented application's reports might get blocked, and eventually lost, if the reporting path used by `liboml2` gets saturated. Similarly, with reports arriving at a rate of more than a few tens of thousands per second, the `oml2-server`'s processing time becomes longer than the inter-packet arrival time. In both cases, the problem can be solved with appropriate planning of the measurement campaign.

First, during the development of an instrumentation, it is important to identify which metrics should be reported together in one MP. For example, it may be relevant to group

fields for which new measurements are produced together (such as various per-packet metrics), but not so much to clump together the readings from multiple heterogeneous sensors.

Second, when preparing the experiment, careful selection of which MPs are selected to generate output, as well as proper use of filtering (performed in the `liboml2` thread) prior to sending the samples over the network, can help reduce the sample rate at the source. The distribution of more than one `oml2-server` or the use of intermediate `oml2-proxy-servers` to stagger the MSs should also be considered.

We direct the reader to [27, 29] for more details.

4.2 Impact on a Wireless Environment

In this section, we present an evaluation of OML's impact when a separate control network is not available to transport the measurement streams. As a result, the measurement collection shares the experimental network with the running experiment, and may disturb it.

4.2.1 Method

We developed a set of experiments to quantify the effect of the OML measurement traffic on the experiment-generated traffic, when they both share the same wireless link. In this case, an impact is indeed expected as both types of traffic contend for the same medium. Thus, we focus on characterising how the experiment traffic varies as a function of the amount of collected measurements.

In that regard, we are interested in two factors. The first one is the **OML sampling rate**, which is the frequency (per sample) at which `trace-oml2` generates an aggregated report and sends it to the server. We vary this factor in the range 1, $\frac{1}{10}$, $\frac{1}{50}$, and $\frac{1}{100}$—the higher the sampling rate, the higher the number of measurements forwarded to the server. Our second factor of interest is the **packet size** that Iperf uses as the MTU for the experimental wireless path. Two values are considered, 1,500 and 1,000 B, in order to explore the impact of a varying number of packets on the same theoretically achievable throughput. We measure a single dependent variable, which is the **achieved throughput** as reported by an OML-instrumented Iperf in legacy mode (same as `nooml` in the previous section) on the receiver.

4.2.2 Results

For this study, we used a wireless testbed similar to ORBIT [42]. The wireless nodes were connected via an 802.11g *ad hoc* network, which carried both experimental and measurement traffic. TCP was used to generate the experimental traffic as this is what OML streams use, thus avoiding any potential TCP/UDP interaction bias.

We focused on Iperf's received TCP rate whilst OML was used within `trace-oml2` and configured to capture information from every packets and send reports on the same wireless network as the Iperf traffic, with varying frequencies. Figure 7 summarises the results, and shows the mean throughputs reported by Iperf as a function of the OML reporting frequency. On this scale, a frequency of 0.01 indicates that the measurement library only reports once every

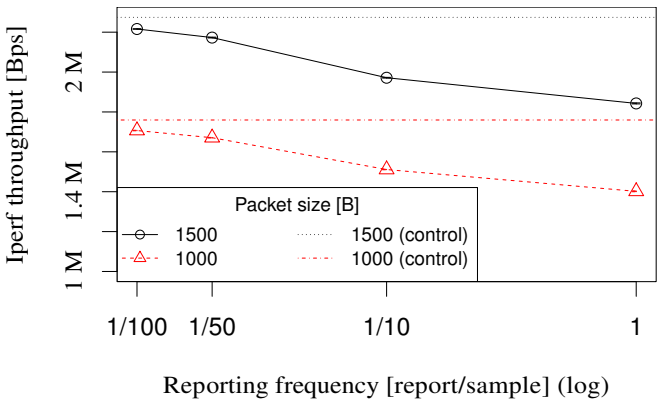

Figure 7: Mean achievable Iperf rate depending on the reporting rate of OML on the same medium (error bars showing the standard error are not visible due to the y-axis scale in use). Factors labelled "control" are the reference Iperf throughput when reporting over a control network.

hundred captured packets. This still represents many reports per seconds. Both treatments 1,500 and 1,000 B of the packet sizes are shown. The behaviour for both packet sizes is qualitatively similar. The figure also includes Iperf's upper-bounds throughput in our wireless environment, measured with `trace-oml2` using a control network rather than the experimental radio network.

The results shown in Figure 7 confirm that the reporting traffic from the OML-instrumented applications impacts the behaviour of the different experiments in a significant manner. This is consistent with the behaviour of two TCP flows sharing the same network, but obviously not desirable in this case. However, these results also demonstrate that, when OML filters are used—in this case, to achieve differentiated sampling policies—this impact on the experiment can be greatly reduced.

4.2.3 Discussion

When no separate network is available to carry the OML measurement streams (such as in wireless experiments with limited number of available channels), OML reporting significantly disturbs the experiments when it is configured with a high measurement sampling rate. While this is an expected behaviour, our study shows that using upstream processing allows the experimenter to asymptotically approach the performance obtained when a dedicated reporting network is available.

These results demonstrate one of the benefit in using inline processing capabilities such as `liboml2`'s filters. The use of `oml2-proxy-servers` in such cases would however be the best option, for measurement campaigns limited in time, if non-filtered data is desired. In these case, a measurement-based feedback control loop (Req. 8) would be extremely desirable.

5 Conclusion

In this paper, we have argued for a Software-Defined Measurement (SDM) framework for the future Internet, compliant with necessary requirements for sound experiments. We have proposed a generic architecture able to comply with

these requirements, allowing researchers to collect, process and store measurements in a unified way, letting them focus on analysis of the data rather than the tedious and error prone tasks of collecting and preparing it.

We also described our complete reimplementation of the OML tool suite. We have shown how this instrumentation suite caters to the main elements of the SDM framework. It allows the collection of measurements and other metrics from any kind of application (Req. 2, 4 and 5), the injection of metadata to support a rich and documented storage format (Req. 3). Moreover it enables reporting, after processing if need be, to local, centralised or distributed collection servers for storage in a unified format (Req. 6). An evaluation of this suite showed that OML could be used with little or no impact on the system under study with regards to changing the behaviour of applications and hardware (Req. 1).

Furthermore, this same evaluation allowed us to confirm of the validity of the proposed framework, in that a carefully tested, library-provided, injection point code allows the experimenter to trust that the instrumentation will not alter the behaviour of the system under study. It also highlighted the value of processing points for the control and steering of measurement streams. Feedback points (Req. 8), though barely implemented in OML were also shown to be useful in a constrained environment in which the sample rate must be reduced, or observations buffered.

References

[1] *Abstract Syntax Notation One (ASN.1): Specification of Basic Notation*. Recommendation X.680. Geneva, Switzerland: ITU-T SG17, Nov. 2008.

[2] S. Alcock, P. Lorier, and R. Nelson. "Libtrace: A Packet Capture and Analysis Library". In: *SIGCOMM Computer Communication Review* 42.2 (Apr. 2012), pp. 42–48.

[3] A. Arasu et al. *STREAM: The Stanford Data Stream Management System*. Tech. rep. 2004-20. Stanford, CA, USA: Stanford InfoLab, Mar. 2004.

[4] S. Avallone, D. Emma, A. Pescapé, and G. Ventre. "Performance Evaluation of an Open Distributed Platform for Realistic Traffic Generation". In: *Performance Evaluation* 60.1-4 (May 2005). Ed. by W. Bux, pp. 359–392.

[5] S. Avallone et al. "D-ITG Distributed Internet Traffic Generator". In: *QUEST 2004, 1st International Conference on Quantitative Evaluation of Systems*. Ed. by G. Franceschinis, J.-P. Katoen, and M. Woodside. University of Twente. Enschede, Netherlands: IEEE Computer Society, Sept. 2004, pp. 316–317.

[6] M. Badger. *Zenoss Core 3.x Network and System Monitoring*. Olton, Birmingham: Packt Publishing, May 2011.

[7] N. Bastin et al. "The InstaGENI Initiative: An Architecture for Distributed Systems and Advanced Programmable Networks". In: *Computer Networks* (2014). Ed. by J. P. G. Sterbenz et al. In press.

[8] M. Berman et al. "GENI: A Federated Testbed for Innovative Network Experiments". In: *Computer Networks* (2014). Ed. by J. P. G. Sterbenz et al. In press.

[9] E. Boschi, B. Trammell, L. Mark, and T. Zseby. *Exporting Type Information for IP Flow Information Export (IPFIX) Information Elements*. RFC 5610. Fremont, CA, USA: RFC Editor, July 2009.

[10] C. Brandauer and T. Fichtel. "MINER — A Measurement Infrastructure for Network Research". In: *TridentCom 2009, 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*. Ed. by T. Magedanz and S. Mao. Washington DC, USA: IEEE Computer Society, Apr. 2009, pp. 1–9.

[11] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. "Dynamic Instrumentation of Production Systems". In: *USENIX 2004*. Ed. by A. Arpaci-Dusseau and R. Arpaci-Dusseau. Boston, MA, USA: USENIX Association, June 2004, pp. 15–28.

[12] S. Chandrasekaran et al. "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World". In: *CIDR 2003, First Biennial Conference on Innovative Data Systems Research*. Ed. by M. Stonebraker. VLDB Foundation; ACM SIGMOD. Asilomar, CA, USA: University of Wisconsin—Madison, Jan. 2003.

[13] B. Claise. *Cisco Systems NetFlow Services Export Version 9*. RFC 3954. Fremont, CA, USA: RFC Editor, Oct. 2004.

[14] B. Claise et al. *Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information*. RFC 5101. Fremont, CA, USA: RFC Editor, Jan. 2008.

[15] M. Eisler. *XDR: External Data Representation Standard*. RFC 4506. Fremont, CA, USA: RFC Editor, May 2006.

[16] *Executive Summary: Computer Network Time Synchronization*. Retrieved 2014-03-11. May 2012. URL: http://www.eecis.udel.edu/~mills/exec.html.

[17] M. Gates, A. Tirumala, J. Dugan, and K. Gibbs. *Iperf version 2.0.0*. NLANR applications support, University of Illinois at Urbana-Champaign. Urbana, IL, USA, May 2004.

[18] A. Hanemann et al. "PerfSONAR: A Service Oriented Architecture for Multi-domain Network Monitoring". In: *ICSOC 2005, 3rd International Conference on Service-Oriented Computing*. Ed. by B. Benatallah, F. Casati, and P. Traverso. Vol. 3826. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer-Verlag Berlin, 2005. Chap. 19, pp. 241–254.

[19] D. Harrington, R. Presuhn, and B. Wijnen. *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*. RFC 3411. Fremont, CA, USA: RFC Editor, Dec. 2002.

[20] M. Huang, A. Bavier, and L. Peterson. "PlanetFlow: Maintaining Accountability for Network Services". In: *ACM SIGOPS Operating Systems Review* 40.1 (Jan. 2006). Ed. by J. N. Matthews and M. E. Fiuczynski, pp. 89–94.

[21] F. Huici et al. "Blockmon: A High-Performance Composable Network Traffic Measurement System". In: *SIGCOMM 2012, Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Demonstration Session*. Ed. by V. Padmanabhan and G. Varghese. Aalto University, Nokia. Helsinki, Finland: ACM, Aug. 2012, pp. 79–80.

[22] G. Iannaccone. *CoMo: An Open Infrastructure for Network Monitoring — Research Agenda*. Tech. rep. Cambridge, UK: Intel Research, Feb. 2005.

[23] G. Jourjon, T. Rakotoarivelo, and M. Ott. "A Portal to Support Rigorous Experimental Methodology in Networking Research". In: *TridentCom 2011, 7th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*. Ed. by H. Li, T. Korakis, P. Tran-Gia, and H.-S. Park. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering. ICST. Shanghai, China: Springer-Verlag Berlin, Apr. 2011.

[24] M. Kim, R. Sumbaly, and S. Shah. "Root Cause Detection in a Service-oriented Architecture". In: *ACM SIGMETRICS Performance Evaluation Review* 41.1 (June 2013). Ed. by J. Douceur and J. Xu, pp. 93–104.

[25] S. S. Kolahi, S. Narayan, D. Nguyen, and Y. Sunarto. "Performance Monitoring of Various Network Traffic Generators". In: *UKSim 2011, 13th International Conference on Computer Modelling and Simulation*. Ed. by R. Cant. Cambridge, United Kingdom: IEEE Computer Society, Mar. 2011, pp. 501–506.

[26] J. a. P. Martins. "Testbed Management Systems". MA thesis. Aveiro, Portugal: Universidade de Aveiro, 2011.

[27] O. Mehani, G. Jourjon, and T. Rakotoarivelo. "A Method for the Characterisation of Observer Effects and its Application to OML". In: (May 2012). arXiv: 1205.3846.

[28] O. Mehani, G. Jourjon, T. Rakotoarivelo, and M. Ott. "An Instrumentation Framework for the Critical Task of Measurement Collection in the Future Internet". In: *Computer Networks* (2014). Ed. by J. P. G. Sterbenz et al. In press.

[29] O. Mehani et al. *Characterisation of the Effect of a Measurement Library on the Performance of Instrumented Tools*. Tech. rep. 4879. Sydney, Australia: NICTA, May 2011.

[30] O. Mehani et al. *Detailed Specifications for First Cycle Ready*. Deliverable FP7-ICT-2011-8-318389-Fed4FIRE/D6.1. EC Information Society Technologies Programme, Mar. 2013.

[31] P. Mell and T. Grance. *The NIST Definition of Cloud Computing*. NIST Special Publication 800-145. Gaithersburg, MD, USA: National Institute of Standards and Technology, Sept. 2011.

[32] H. Niavis et al. *Wireless-specific Measurement Framework*. Deliverable OpenLab/D3.4. EC Information Society Technologies Programme, Aug. 2013.

[33] R. Olups. *Zabbix 1.8 Network Monitoring*. Olton, Birmingham: Packt Publishing, 2010.

[34] Open Networking Foundation. *Software-Defined Networking: The New Norm for Networks*. White paper. Palo Alto, CA, USA: Open Networking Foundation, Apr. 2012.

[35] K. Park and V. S. Pai. "CoMon: A Mostly-Scalable Monitoring System for PlanetLab". In: *ACM SIGOPS Operating Systems Review* 40 (Jan. 2006). Ed. by J. N. Matthews and M. E. Fiuczynski, pp. 65–74.

[36] V. Paxson. "Strategies for Sound Internet Measurement". In: *IMC 2004, 4th ACM SIGCOMM Conference on Internet Measurement*. Ed. by J. Kurose. Taormina, Sicily, Italy: ACM, Oct. 2004, pp. 263–271.

[37] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, May 2010.

[38] M. Rabinovich, S. Triukose, Z. Wen, and L. Wang. "DipZoom: The Internet Measurements Marketplace". In: *INFOCOM 2006, Proceedings of 25th IEEE International Conference on Computer Communications, Global Internet Symposium*. Ed. by A. Azcorra, J. Touch, and Z.-L. Zhang. Barcelona, Spain: IEEE, Apr. 2006, pp. 1–6.

[39] A. Rajasekar et al. "iRODS Primer: Integrated Rule-Oriented Data System". In: *Synthesis Lectures on Information Concepts, Retrieval, and Services* 2.1 (Jan. 2010), pp. 1–143.

[40] T. Rakotoarivelo, G. Jourjon, and M. Ott. "Designing and Orchestrating Reproducible Experiments on Federated Networking Testbeds". In: *Computer Networks* (2014). Ed. by J. P. G. Sterbenz et al. In press.

[41] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar. "OMF: A Control and Management Framework for Networking Testbeds". In: *ACM SIGOPS Operating Systems Review* 43.4 (Jan. 2010). Ed. by M. E. Fiuczynski and J. Matthews, pp. 54–59.

[42] D. Raychaudhuri et al. "Overview of the ORBIT Radio Grid Testbed for Evaluation of Next-Generation Wireless Network Protocols". In: *WCNC 2005, IEEE Wireless Communications and Networking Conference*. Ed. by K. C. Chua et al. Vol. 3. New Orleans, LA, USA: IEEE, Mar. 2005, pp. 1664–1669.

[43] L. Sanchez et al. "SmartSantander: IoT Experimentation over a Smart City Testbed". In: *Computer Networks* (Dec. 2013). Ed. by J. P. G. Sterbenz et al.

[44] F. Schneider, J. Wallerich, and A. Feldmann. "Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware". In: *PAM 2007, 8th Internatinal Conference on Passive and Active Network Measurement*. Ed. by S. Uhlig, K. Papagiannaki, and O. Bonaventure. Vol. 4427. Lecture Notes in Computer Science. Louvain-la-neuve, Belgium: Springer-Verlag Berlin, Apr. 2007. Chap. 21, pp. 207–217.

[45] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889. Fremont, CA, USA: RFC Editor, Jan. 1996.

[46] U. Sedlar et al. “Contextualized Monitoring and Root Cause Discovery in IPTV Systems Using Data Visualization”. In: *IEEE Network* 26.6 (Nov. 2012). Ed. by X. S. Shen, pp. 40–46.

[47] Y. Shavitt and E. Shir. “DIMES: Let the Internet Measure Itself”. In: *SIGCOMM Computer Communication Review* 35.5 (Oct. 2005), pp. 71–74.

[48] M. Singh, M. Ott, I. Seskar, and P. Kama. “ORBIT Measurements Framework and Library (OML): Motivations, Design, Implementation, and Features”. In: *TridentCom 2005, 1st International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*. Ed. by J. Aracil, S. Kalyanaraman, and K. Mase. Trento, Italy: IEEE Computer Society, Feb. 2005, pp. 146–152.

[49] D. Veitch, J. Ridoux, and S. B. Korada. “Robust Synchronization of Absolute and Difference Clocks over Networks”. In: *IEEE/ACM Transactions on Networking* 17.2 (Apr. 2009). Ed. by D. Towsley, pp. 417–430.

[50] Z. Wen, S. Triukose, and M. Rabinovich. “Facilitating Focused Internet Measurements”. In: *SIGMETRICS 2007, international conference on Measurement and modeling of computer systems*. Ed. by M. Ammar and M. Harchol-Balter. San Diego, CA, USA: ACM, 2007, pp. 49–60.

[51] J. White. *OML User Manual*. OML — The OMF Measurement Library. Work in progress, included in the OML source distribution (for version 2.4.0). NICTA. Alexandria, NSW, Australia, Oct. 2009.

[52] J. White, G. Jourjon, T. Rakotoarivelo, and M. Ott. “Measurement Architectures for Network Experiments with Disconnected Mobile Nodes”. In: *TridentCom 2010, 6th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*. Ed. by A. Gavras, N. Huu Thanh, and J. Chase. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering. ICST. Berlin, Germany: Springer-Verlag Berlin, May 2010.

[53] M. Yu, L. Jose, and R. Miao. “Software DeïňĄned TrafiňĄc Measurement with OpenSketch”. In: *NSDI 2013, 10th USENIX Symposium on Networked Systems Design and Implementation*. Ed. by N. Feamster and J. Mogul. USENIX. Berkeley, CA, USA: USENIX Association, Apr. 2013, pp. 29–42.

[54] Q. Zhao, Z. Ge, J. Wang, and J. Xu. “Robust Traffic Matrix Estimation with Imperfect Information: Making Use of Multiple Data Sources”. In: *SIGMETRICS Performance Evaluation Review* 34.1 (June 2006). Ed. by P. Key and E. Smirni, pp. 133–144.

A Code Generation with the `oml2-scaffold(1)` Helper

Along with the developer tools of `liboml2` comes `oml2-scaffold`. This helper utility is a code generator based on a high-level description of an application and its measurement points (see Listing 4).

```
# example.rb, to be processed with oml2-scaffold(1)
defApplication('oml:app:example', 'example') do |a|
  a.version(1, 0, 0)
  a.shortDescription = 'Example OML application'
  a.description = 'Example OML description for
      oml2-scaffold'

  a.defMeasurement("example_mp") do |m|
    m.defMetric('label', :string)
    m.defMetric('n', :uint32)
  end

  a.path = "/usr/local/bin/example"
end
```

Listing 4: A high-level application description, written in a dialect of Ruby, that `oml2-scaffold` can use to generate code for the instrumentation helpers, as well as the full skeleton of the application, if need be.

The most important code block is the header (`example_oml.h` in Listing 2) declaring the MPs as well as helper functions for their registration and injection using `liboml2`'s low-level client functions (`omlc_*()` functions).

However, `oml2-scaffold` can also generate a full skeleton for the application, when it is written from scratch, including the command-line parsing mechanism, and the Makefile to build the binary. It can also generate an example application description (such as the one in Listing 4), to ease the bootstrapping of new instrumentations.

Moreover, this description can be used directly by OMF [40, 41] to deploy and configure the instrumented application as part of an experiment.

B Writing Wrappers for External Tools

When the code of an application to instrument is not available, or when it is not convenient to modify it, it is possible to write wrappers to take care of the OML injection. The OML4R and OML4Py bindings are well adjusted for this task. As an example, Listing 5 shows how to use OML4R to capture the output of the `ping` command, parse it using a regular expression, and report the measured round-trip times over OML.

A more fully-fledged version is available alongside the OML4R gem and can be perused at http://oml.mytestbed.net/projects/oml/repository/oml4r/entry/bin/ping-oml2?rev=release%2F2.10.

```ruby
#!/usr/bin/ruby
# OML4R wrapper for ping
# This application runs the system ping, parses its
    output and reports the
# measurements via OML
require 'rubygems'
require 'oml4r'

# Declare the measurement point
class MPStat < OML4R::MPBase
  name :ping
  param :dest_addr, :type => :string
  param :ttl, :type => :uint32
  param :rtt, :type => :double
  param :rtt_unit, :type => :string
end

# Parse the command line (including standard --oml-X
    options)
inet6 = ''
addr = nil
leftover = OML4R::init(ARGV, :appName => 'ping') do
    |argParser|
  argParser.banner = "Runs the system ping and
      reports measurements via OML\n"
  argParser.on("-6","--[no]-inet6 ","Use ping6 rather
      than ping") { @inet6 = "6"}
end
addr = leftover[0]

# Run the appropriate ping(8) tool, parse its output
    and report over OML
IO.popen("/bin/ping#{inet6} #{addr}") do |pipe|
  while row = pipe.gets do
    if not (parse =

        /^\d+ bytes from (?<host>.*):
        icmp_seq=(?<icmp_seq>\d+) ttl=(?<ttl>\d+)
        time=(?<time>[0-9.]+)
        (?<timeunit>[a-zA-Z]+)/.match(row)
      ).nil? # One sample
      MPStat.inject(parse[:host], parse[:ttl],
          parse[:time], parse[:timeunit])
    end
  end
end
OML4R::close
```

Listing 5: An OML4R-based wrapper for the `ping` command. The output is parsed in a loop using the Ruby regular expression syntax. Each extracted sample is then injected into the reporting chain.