AURIC: A Scalable and Highly Reusable SLA Compliance Auditing Framework

Hasan, Burkhard Stiller

Computer Science Department IFI, University of Zürich, Switzerland
[hasan|stiller]@ifi.uzh.ch

Abstract. Service Level Agreements (SLA) are needed to allow business interactions to rely on Internet services. Service Level Objectives (SLO) specify the committed performance level of a service. Thus, SLA compliance auditing aims at verifying these commitments. Since SLOs for various application services and end-to-end performance definitions vary largely, *automated* auditing of SLA compliances poses the challenge to an auditing framework. Moreover, end-to-end performance data are potentially large for a provider with many customers. Therefore, this paper presents a *scalable* and *highly reusable* auditing framework and a prototype, termed *AURIC* (***Au***diting F***r***amework for ***I***nternet Servi***c***es), whose components can be distributed across different domains.

1 Introduction

Today, the Internet has become a platform for business. Various services are offered to enable business transactions to be accomplished. A *Service Level Agreement* (SLA) is negotiated between a provider and a customer in order to define a legally binding contract regarding the service delivery. While the TeleManagement Forum defines an SLA as "a formal negotiated agreement between two parties, sometimes called a Service Level Guarantee, it is a contract (or part of one) that exists between the service provider and the customer, designed to create a common understanding about services, priorities, responsibilities, etc." [17], in general, an SLA comprises in particular a service description, the *expected performance level* of the service, the procedure for reporting problems, the *time-frame for response and problem resolution*, the process for monitoring and reporting the service level, the consequences for the provider not meeting its obligations, and escape clauses and constraints [18]. The performance level of a service committed is specified in a set of *Service Level Objectives* (SLO). Thus, SLA compliance auditing aims at verifying that these SLOs are met. This task must be *automated* in order to be *efficient* and to enable *real-time reactions* in case of an SLA violation.

In fact, specifying SLAs on IP-based networks becomes viable through network device instrumentations for Quality-of-Service (QoS) measurements, not only of transport but also of application services. However, application service SLAs still pose challenges to their compliance auditing, due to the *variety* and the potential *complexity* of SLOs. An example for a complex SLO is the following detail specification of service availability: "In most cases, service requests from authorised users will be accepted. If a request from an authorised user is rejected or not responded within 15 seconds, then the next request for this service from the same user will be accepted. However, this next request must be made within the next 5 minutes and 1 minute must have been elapsed since the rejected or unresponded request." Thus, an *expressive* specification language is beneficial to formally specify such complex relations among various events.

A useful auditing framework must allow for the *distribution of auditing load* to separate auditor instances. The time and memory required for auditing may increase only *linearly* with an increasing number of audit data. Moreover, the framework must be *reusable* and *easily* adaptable to audit any *complex* SLO. Hence, this paper presents a *scalable* and *highly reusable generic* framework, termed *AURIC* (***Au***diting F***r***amework for ***I***nternet Servi***c***es), which supports *secure inter-domain* interactions and provides all necessary core functionality to conduct *automatically* potentially *complex* audit tasks.

The remainder of this paper is organized as follows. Section 2 discusses related work. While Section 3 presents the AURIC architecture for SLA compliance auditing, Section 4 describes its prototypical implementation. An extensive evaluation of AURIC with respect to its scalability and reusability is presented in Section 5, which is followed by Section 6, where conclusions are drawn.

2 Related Work

Current approaches in SLA management address the *formal specification* of a complete SLA in a *specific area*, *e.g.*, network or web services, or concentrate on measurements of a pre-defined set of SLA parameters [1], [6], [8], [10], [12], [13]. Hence, to modify or to extend an existing solution, particularly a commercial product, for its application to an SLO with a different logic, a larger effort is needed than if the solution has been based on a generic framework like AURIC. Moreover, most approaches support only *simple SLO terms* and do not consider possible *inter-domain* auditing interactions and their *security* requirements. While [7] discusses all relevant details of related work, the following paragraphs summarize major issues only.

The Web Service Level Agreement (WSLA) Framework proposes a concept for SLA management including online monitoring of SLA violation and defines a language to specify SLAs [13]. However, it *focuses on web services and supports only simple SLO terms*. A condition in a WSLA's SLO is simply a logic expression with SLA parameters as variables. WSLA does not support conditional expressions for SLO specifications and the framework does not expect to process metered data consisting of more than one field, *e.g.*, <IPAddress, PacketLossRatio>. Since the timepoint at which the value of a measured metric is transferred is considered as the measurement timepoint, batch processing of measured data is not supported.

In the area of Grid services, Cremona [14] is an architecture and library for the creation and monitoring of WS-Agreements, whose specification is worked out by the Grid Resource Allocation and Agreement Protocol Working Group (GRAAP-WG) of the Global Grid Forum. Cremona supports the implementation of agreement management, however, SLO monitoring is considered application specific, thus, no support to its implementation is available, except an interface to retrieve monitoring results.

The Project TAPAS (Trusted and Quality-of-Service Aware Provision of Application Services) proposes SLAng, a language for expressing SLAs precisely [16]. SLAng is defined using an instance of the Meta-Object Facility model, and its violation semantics is defined using Object Constraint Language constraints. To reduce the possibility of disagreement over the amount of errors introduced by the mechanism for SLA violation detection, a contract checker is to be automatically generated by using the metamodel of the language and associated constraints as inputs for a generative programming tool [15]. However, this approach leads to performance problems. Thus, in order to eliminate various drawbacks mentioned above, this paper presents an architecture for SLA compliance auditing as described in the next section.

3 AURIC SLA Compliance Auditing Architecture

Based on the generic model and architecture for automated auditing [9], the AURIC architecture for SLA compliance auditing has been implemented, which covers three main functions: metering, accounting, and auditing, as depicted in Fig. 1.

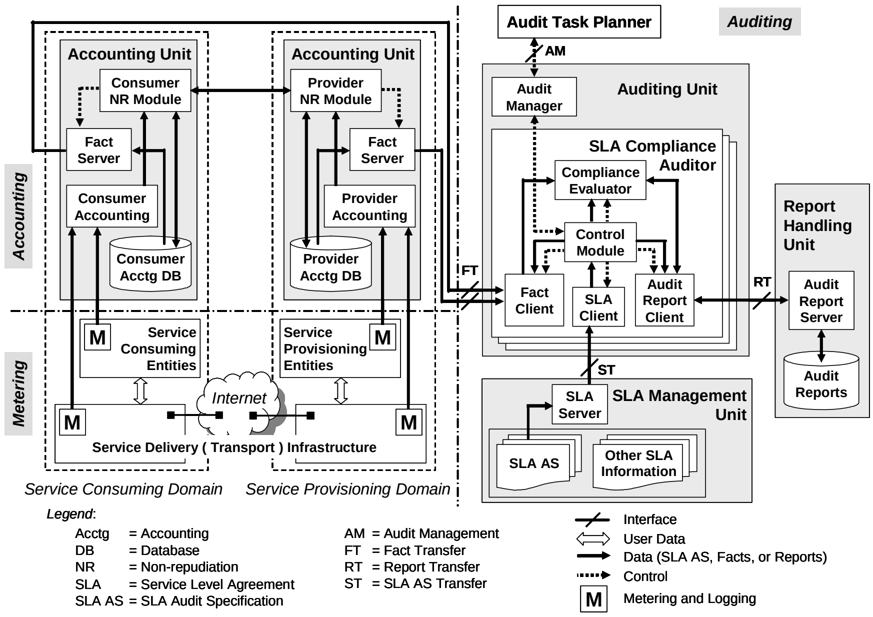


Fig. 1. AURIC SLA Compliance Auditing Architecture

Metering and Accounting: The quality level of a service being delivered must be metered to allow for the auditing of its SLA compliance. Metered data are collected and aggregated by accounting components to generate accounting data (termed Facts). Accounting data are passed to the non-repudiation (NR) module to generate evidence of service consumption. Generation and transfer of evidences require interactions between NR modules from both sides. Accounting data and evidences are stored in the accounting database and the respective Fact server is notified, so that they are transferred to the

SLA compliance auditor. If non-repudiation is not required, an NR module simply acts as a proxy between the accounting component and the database or the Fact server. The architecture and protocols for non-repudiation of service consumption supporting fairness and identity privacy in a mobile environment are available [7], [11].

Auditing: The main interactions between AURIC's components are for auditing. The auditing unit provides an auditing service through the Audit Management (AM) interface. The audit manager waits for audit requests and forwards each audit task received to an auditor. It also accepts requests relating to an audit task being conducted, *e.g.*, requests on its status and requests to stop an audit task. An audit task planner represents an entity which requests an auditing service from an auditing unit. An auditor retrieves data to be processed from various sources: accounting units, SLA management units, and Report handling units. Each of these components provides for a service to access its data through a data server component, namely a Fact server, an SLA server, and an Audit Report server respectively. Note that an Audit Report server also receives requests to store Audit Reports. All SLOs committed are assumed to be specified in a language, which allows for an automated auditing. The resulted specifications are called SLA Audit Specifications (SLA AS). Other SLA information, *e.g.*, user profile, service profile, are not relevant at this stage, and thus, are not explicitly listed in the figure.

To communicate with various data servers, an auditor must contain the corresponding clients. The communication happens via the respective interface: SLA AS Transfer (ST), Fact Transfer (FT), or Report Transfer (RT) interface. The auditor must also contain a compliance evaluator to examine accounting data and Audit Reports based on the SLA AS obtained from the SLA client through the control module. The control module configures and controls other components in carrying out their functions. The Fact client retrieves accounting data and delivers them to the compliance evaluator. If needed, the Audit Report client retrieves and delivers Audit Reports to the compliance evaluator. Finally, this client sends Audit Reports obtained from the compliance evaluator to a Report handling unit. Table 1 briefly discusses suitable protocols for those interfaces.

Table 1. Auditing Interfaces

Interface	Description
AM	A new protocol for this interface is needed, however, following two communication patterns are sufficient to enable management interactions in normal and erroneous situations: Request-Answer and Notification pattern. A request message is used to initiate or terminate an audit task or to obtain its status information. An answer is sent as a response to a request message and it may contain error description, if any. A notification can be sent at any time to inform the respective audit task planner of completion of an audit task or any error occured during an audit.
ST	A URL is used to locate a particular SLA AS. Existing protocols such as HTTPS and SSH File Transfer Protocol are very well suited to be used to transfer SLA AS securely from an SLA manager to the auditing unit.
FT	For the purpose of transferring Facts, Diameter [2] protocol is very well suitable. The Base Accounting message pair is sufficient. However, to allow for selection of Facts a new Diameter command must be defined.
RT	Diameter is also suitable here, since the types of interactions are the same as for FT interface.

Security Considerations: Multi-domain support requires secure interactions and access control. Since in an SLA all parties involved are known in advance, security associations among those components can be established before interactions take place.

Having these security associations in place, authentication and authorization (AA) can be accomplished. As an example, suppose that accounting unit and auditing unit are located in different administrative domains. There are several ways of doing access control, *e.g.*, based on Authentication, Authorization, and Accounting (AAA) architecture [3]. In Fig. 2 (a), an AA server is contacted by the accounting unit to authenticate and authorize the auditing unit before it is allowed to send data to the auditing unit.

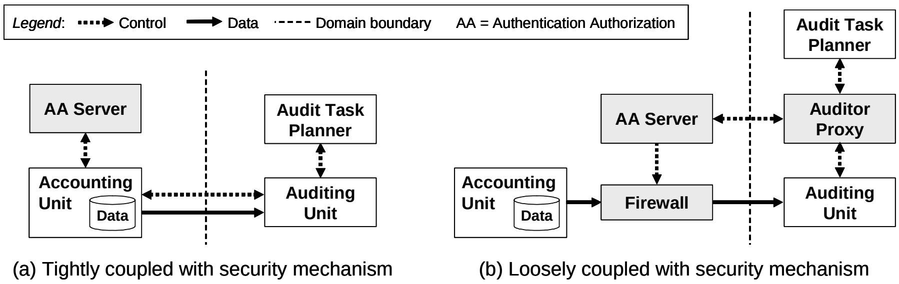


Fig. 2. Examples of Secured Access to Accounting Data

Access control can also be provided without intervening auditing functionality as shown in Fig. 2 (b). An auditor proxy is inserted between audit task planner and auditing unit. The proxy analyses audit tasks and requests access to the relevant accounting unit from the AA server of the respective domain. If there is a security association between the two domains, the access request is accepted and the firewall is configured to allow data flows between the auditing unit and the accounting unit. On receipt of a positive response from the AA server, the proxy forwards the audit task to the auditing unit. Finally, if necessary, a secure communication channel can be established to transfer data confidentially, based on security associations between those domains.

4 Implementation

Based on the proposed architecture, a prototypical implementation of an SLA compliance auditing framework in C++ is provided. The implementation aims at showing that developing an auditor can be done basically through specialization of a set of base classes to implement the SLO specific application logic. Fig. 3 depicts the implementation architecture of a *specific* SLA compliance auditor. The auditor is specific, since the application logic to audit a *specific SLO* is implemented as an integral part of the auditor. Thus, the auditor does not require an SLA client component to retrieve the SLA AS (cf. Fig. 1). However, various application logic corresponding to different SLOs can be implemented at compile time before one is chosen to be applied through a configuration file at run time. Thus, the need of a parser.

Each Fact and Audit Report is represented as a list of attribute-value-pairs (AVPs). Diameter [2] is chosen as the protocol for transferring Facts and Audit Reports due to its extensibility and the capability of its accounting message to carry a list of AVPs. Thus, the functionality of a Fact client and a Report client (cf. Fig. 1) is merged into a single entity called a Fact and Report client, which consists of a Fact and Report transfer

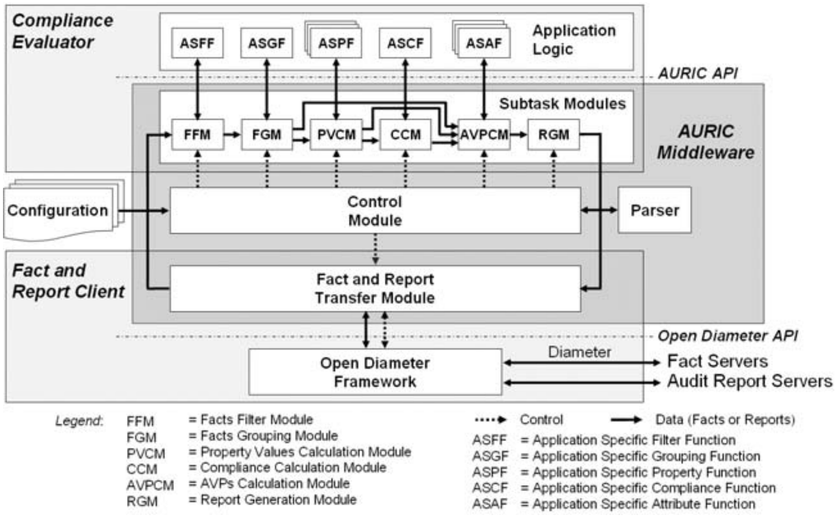


Fig. 3. Implementation Architecture of an SLA Compliance Auditor

module implemented on top of the Open Diameter framework. The description of the Open Diameter implementation is given in [5]. Furthermore, to obtain a modular design, the author proposes to decompose an audit task into a sequence of subtasks:

1. ***Facts filtering***: Only Facts which are relevant for the SLO being audited are to be further processed. The filtered Facts are named *related Facts*.
2. ***Facts grouping***: Related Facts must be grouped, since they result from different service settings or observation periods. A group of Facts from a particular setting or period is named a *Fact-List*. A Fact-List being built is called an *open* Fact-List, whereas a Fact-List ready for auditing is called a *complete* Fact-List.
3. ***Property values calculation***: Each performance parameter of a service is characterized by a set of *properties*, whose values are calculated from the complete Fact-List examined in order to determine the compliance with the SLO.
4. ***Compliance calculation***: The *degree of compliance* with the SLO is calculated by applying the SLO specific compliance formula to the property values.
5. ***Report AVPs calculation***: The values for the report AVPs are calculated from various sources: the Fact-List, property values, and the compliance value.
6. ***Report generation***: As a result, a report is composed from the report AVPs.

Based on this decomposition, the compliance evaluator is developed, which consists of two parts: a sequence of subtask modules and a set of application logic. While the application logic implements SLO specific subtask functions, the subtask modules implement functionality which is common to all auditing applications, namely, management of Facts, Fact-Lists, and property values, as well as transfer of data between two subtask modules. The interface between a subtask module and its application logic is defined by the AURIC Application Programming Interface (API).

4.1 AURIC API

The auditing framework API provides five base classes to implement application logic (cf. Fig. 4). The parent class `SubtaskFunc` provides methods to parametrize the application specific subtask function derived, which are invoked by the auditing framework after the creation of the function based on the configuration file. Each base class offers a method `Process()`, whose purpose is described in Table 2 and which should be implemented by the developer of the auditing application.

Table 2. The Purpose of the API's `Process()` Methods.

Class	The Purpose of `Process()` Method
`Filter-Function`	To examine the accounting record encapsulated in the `Fact` object and return true or false to denote whether the record is related to the SLO being audited. A `Fact` object provides for methods to get information about the accounting record encapsulated in the object, *e.g.*, the value of a particular attribute.
`Grouping-Function`	To examine the accounting record encapsulated in the `Fact` object and assign the record to one or more Fact-Lists with the help of `OpenFactLists` object. An `OpenFactLists` object provides for methods to manipulate open Fact-Lists managed by the auditing framework, *e.g.*, to add a Fact into an open Fact-List and to close an open Fact-List.
`Property-Function`	To calculate a property value from the list of related accounting records encapsulated in the `FactList` object. A `FactList` object provides for methods to manipulate and to access information about accounting records encapsulated in the object, *e.g.*, the number of records, the sum of the value of a particular field of the records.
`Compliance-Function`	To calculate a compliance value from the list of property values encapsulated in the `PropertyValues` object. A `PropertyValues` object provides for methods to access property values.
`Attribute-Function`	To calculate a report attribute value from the list of related accounting records (encapsulated in `FactList` object), the list of property values (encapsulated in the `PropertyValues` object), and the compliance value.

4.2 Development of a General SLA Compliance Auditor

A *general* SLA compliance auditor is an auditor which can be used to audit *any* SLO without the need to modify and recompile the application logic. To implement a general SLA compliance auditor, following items must be available: an audit specification *language* to define in detail *how* an SLO is to be audited and an implementation of those five application specific classes as an *interpreter* of the audit specification language used. An audit specification language, named *Sapta*, has been developed [7].

A Sapta specification for auditing an SLO consists of a set of *function definition subspecifications* and a set of *function invocation subspecifications*. Each set of function definition subspecifications defines the application logic corresponding to those five functions defined in Section 4.1 to audit a specific SLO, whereas each set of function invocation subspecifications defines which function definition subspecifications are to be invoked and with which values for their parameters. The function invocation subspecifications in Sapta is usable as a configuration file for auditing, which consists of a `ComplianceCalculation` subspecification and a `ReportComposition` subspecification. Furthermore, the following principle is followed in the design of Sapta: The management (storage and transport) of Facts and Fact-Lists should be transparent to a pro-

```
class SubtaskFunc {
 public:
  virtual ~SubtaskFunc() {}
  virtual bool SetStringParam(
   unsigned int paramNo,
   const string& paramVal) {return false;}
  virtual bool SetNumberParam(
   unsigned int paramNo,
   float paramVal) {return false;}
  virtual bool SetBooleanParam(
   unsigned int paramNo,
   bool paramVal) {return false;}
};
class FilterFunction : public SubtaskFunc {
 public:
  virtual ~FilterFunction() {}
  virtual bool Process(
   const Fact& currentFact) = 0;
};
class GroupingFunction : public SubtaskFunc {
 public:
  virtual ~GroupingFunction() {}
  virtual void Process(const Fact& currFact,
   OpenFactLists& ofl) = 0;
};

class PropertyFunction : public SubtaskFunc {
 public:
  virtual ~PropertyFunction() {}
  virtual prop_value_t* Process(
   FactList& currentFactList) = 0;
};

class ComplianceFunction: public SubtaskFunc {
 public:
  virtual ~ComplianceFunction() {}
  virtual float Process(
   const PropertyValues& propertyValues) = 0;
};

class AttributeFunction : public SubtaskFunc {
 public:
  virtual ~AttributeFunction() {}
  virtual void Process(string& attrValue,
   FactList& currentFactList,
   const PropertyValues& propertyValues,
   float complianceValue) = 0;
};
```

Fig. 4. AURIC API

grammer of an audit specification. Accesses to and manipulations of Facts and Fact-Lists are to be supported through specific language constructs. Thus, in addition to conventional language constructs such as iteration and conditional branches, Sapta defines constructs which allow for a convenient specification of audit subtasks, *e.g.*, time schedule to evaluate completeness of a Fact-List (cf. Chapter 4 in [7] for further details).

5 Evaluation

The AURIC framework is evaluated with respect to its key requirements defined in Section 1. The scalability of the architecture is analyzed with respect to the number of SLOs, while the load scalability of its implementation, in terms of processing delay and memory requirements, is evaluated with respect to the number of Facts to be processed.

5.1 Scalability of Auditing Framework

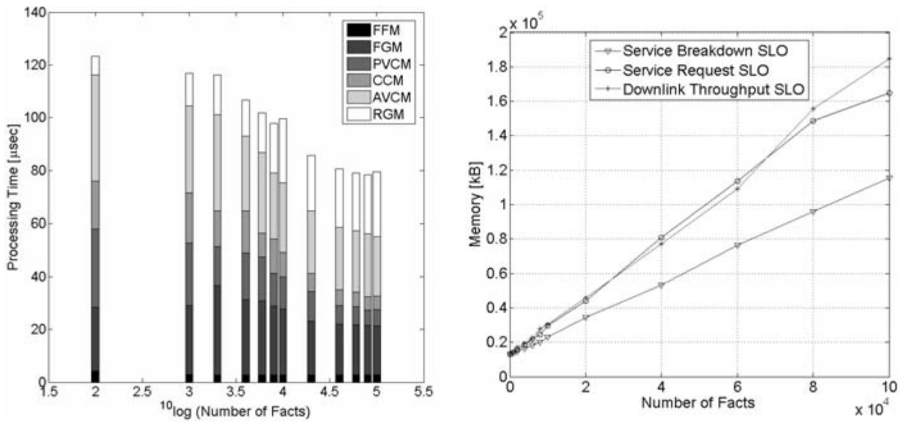
Suppose that there are p parties in a multi-domain environment and two SLAs are negotiated between any two parties (in one SLA a party takes the role of a service provider, in the other SLA the role of a customer). This full mesh relationship results in $p*(p-1)$ SLAs. However, from the point of view of each party only $2*(p-1)$ SLAs are relevant. Unlike other approaches which use an auditor instance per SLA, AURIC defines an auditor instance per SLO. The number of SLOs (n_{SLO}) does not depend on the number of SLAs (n_{SLA}), but on the number of services (n_{svc}). Assuming that each service has a maximum of c SLOs, then n_{SLO} is bound by $c*n_{svc}$. Table 3 compares the scalability of AURIC architecture with the other approaches, where n_A is the number of auditor instances required and $n_{A,max}$ is its upper bound. Although all approaches show a *linear scalability*, AURIC does have an *advantage* over the other: the number of services and SLOs grows much slower than the number of customers (SLAs).

With respect to the load scalability of an auditor, the number of Facts to be audited is crucial. There is a limit to the processing speed of an auditor, which determines the amount of Facts allowed per time unit. The amount of Facts can increase due to, *e.g.*, more sessions, which are generated. By *scaling up* the auditor, more Facts can be audited. However, this problem can also be solved by *scaling out* the auditor, since accounting data for the same SLO can be partitioned (*e.g.*, based on CustomerID) and delivered to several instances of auditors, all responsible for the same SLO.

Table 3. Scalability Comparison

Approach	n_A	$n_{A,max}$	Order of n_A
WSLA Framework, Cremona, TAPAS SLAng	n_{SLA}	$2 * (p-1)$	O(n)
AURIC	n_{SLO}	$c * n_{svc}$	O(n)

Auditor Processing Time: To evaluate the processing time, three SLO specific auditors are implemented based on the AURIC framework. Each auditor is responsible for auditing one of the three SLOs: Service Breakdown SLO, Service Request SLO, and Downlink Throughput SLO. The measurement of the processing time is done on a host with a Pentium 4 CPU 1.80 GHz, 512 MB main memory. Facts to be processed are delivered at once in a single batch to the auditor, and experiments are carried out with different numbers of Facts. In each experiment the time needed by those Facts to pass processes from the first to a certain subtask module is measured. Each experiment is run 10 times with the same configuration to obtain an average value of the processing time. For example, results show that it takes *in average* 7.94 s (with a *standard deviation* of 0.16 s) to process 100,000 Facts delivered at once through the sequence of *all* subtask modules in auditing the service breakdown SLO.



(a) Average Processing Time per Fact (b) Heap Memory Usage

Fig. 5. Load Scalability

Fig. 5 (a) depicts as an example the average processing time *per Fact* in *each* subtask module for auditing service breakdown SLO. Other use cases see similar results. The time required by an auditor to accomplish its task is determined by the total number of Facts to be processed, the number of *related* Facts after being *filtered*, the number of

Fact-Lists after being *grouped*, and the *complexity* of the SLO defined. In all use cases, for a large number of Facts the processing time per Fact in each subtask module exhibits a relative constant value as expected. Thus, AURIC shows a scalable implementation.

Auditor Heap Memory Usage: Memory requirements of the auditor are important, especially in relation to the number of Facts. Hence, for those three use cases the memory usage is obtained from /proc files [4]. The virtual memory usage of the heap determines the dominating aspect, thus, all other memory usage is omitted. If all Facts are delivered *at once* to the auditor, a *linear* increase of heap memory usage with an increasing number of Facts is expected, since more memory will be needed to store more Facts. This behavior is shown in Fig. 5 (b), showing that the AURIC implementation scales.

5.2 Reusability of Auditing Framework

High reusability is a very important property to be fulfilled by an auditing framework. AURIC's reusability is shown by demonstrating that most of the auditing components do not need to be adapted or replaced, when developing a new auditing application based on the framework. Assuming the example of the following application logic to determine compliances of Facts with a certain SLO:

- If a `Fact` belongs to the SLO to be audited then `ff1(Fact)` is true.
- The value of `gf1(Fact, OpenFactLists)` identifies the `FactList` to which the `Fact` belongs (*e.g.*, all accounting records about (un)availability of service S within a month are to be grouped in order to decide on SLO compliance). If a `FactList` is complete, then `gf2(FactList)` is true.
- A `FactList` complies with the SLO if the value of `cf1(pf1(FactList), pf2(FactList))` is 1 (*e.g.*, if service S may down at most 3 times which are longer than 5 minutes, and the total downtime may not exceed 30 minutes, then `pf1()` would count the number of breakdowns longer than 5 minutes and `pf2()` would calculate the total downtime).
- If a `FactList` does not comply with the SLO a report consisting of `pf1(FactList)`, `pf2(FactList)`, `af1(FactList)`, and `cf1(pf1(FactList), pf2(FactList))` is to be generated.

This logic is easily implemented into AURIC by writing those five application-specific functions. Fig. 6 depicts the *simplified* code snippets. Having defined these subclasses, the programming job is done and an executable auditor for this specific SLO can be compiled. All other functionality is provided automatically by the framework, *e.g.*, interactions with Fact/Report servers to obtain Facts and to deliver Audit Reports, management of Facts, Fact-Lists, property values, and execution of methods invoked by audit subtasks, as well as transfer of data between audit subtasks.

Before invoking the newly developed auditor, a configuration file written in Sapta needs to be created. The framework consults this file to determine, which subclasses are to be used by each audit subtasks and to determine the composition of an Audit Report. For the example above, the content of the configuration file is shown in Fig. 7. Furthermore, it is likely that several SLOs share the same application logic for specific functions, *e.g.*, a `PropertyFunction` to determine the average value of a certain field in the accounting records. This subclass needs to be coded once and can be used for various SLOs through auditor configurations. Thus, the framework also supports reuse of application logic without code duplication in addition to the reuse of its own components.

```
class FF_SLO1 : public FilterFunction {
 public:
  bool Process(const Fact& currentFact)
    {return ff1(currentFact);}
};
class GF_SLO1 : public GroupingFunction {
 public:
  void Process(const Fact& currentFact,
               OpenFactLists& ofl) {
    thisFactListId = gf1(currentFact, ofl);
    ofl.Assign(thisFactListId, currentFact);
    if (gf2(ofl.GetFactList(thisFactListId)))
     {ofl.CloseFactList(thisFactListId);}
  }
};
class PV_SLO1 : public prop_value_t {
// define variables to store a property value
};
class PF_1_SLO1 : public PropertyFunction {
 public:
  prop_value_t* Process(FactList& currFL) {
    PV_SLO1* pv = new PV_SLO1;
    // assign pf1(currFL) to variables in pv
    return ((prop_value_t*)pv);
  }
};
class PF_2_SLO1 : public PropertyFunction {
 public:
  prop_value_t* Process(FactList& currFL) {
    PV_SLO1* pv = new PV_SLO1;
    // assign pf2(currFL) to variables in pv
    return ((prop_value_t*)pv);
  }
};
class CF_SLO1 : public ComplianceFunction {
 public:
  float Process(const PropertyValues& pVal) {
    PV_SLO1& pv1 = (PV_SLO1&)
      pVal.GetPropertyValue(1);
    PV_SLO1& pv2 = (PV_SLO1&)
      pVal.GetPropertyValue(2);
    return (cf1(pv1, pv2));
  }
};
class AF_SLO1 : public AttributeFunction {
 public:
  void Process(string& attrValue,
      FactList& currentFactList,
      const PropertyValues& propertyValues,
      float complianceValue) {
    attrValue = af1(currentFactList);
  }
};
```

Fig. 6. Deriving Application Specific Functions

```
ComplianceCalculation CC_SLO1 {
        FF_SLO1
        >> GF_SLO1
        >> PF_1_SLO1, PF_2_SLO1
        >> CF_SLO1
}
ReportComposition RC_SLO1 {
        [Field1 eq GF_SLO1 >> AF_SLO1],
        [Field2 eq PF_1_SLO1],
        [Field3 eq PF_2_SLO1],
        [Field4 eq CF_SLO1]
}
```

Fig. 7. Example Configuration in Sapta

6 Summary and Conclusions

Existing approaches in SLA compliance auditing lack a *general applicability* and concentrate on formal specifications of SLAs rather than on the auditing of SLOs. These pure specification approaches lead to the potential unawareness of system designers on how manifold and complex an SLO for application services can be beyond a guarantee of traditional QoS parameters. Thus, AURIC has been designed based on a *generic model and architecture*. Since the architecture neither assumes specific services nor specific SLOs, it is *general* and applicable to the full range of Internet service types. Furthermore, AURIC architecture is shown to be *linearly scalable* with respect to the number of SLOs due to the possibility to employ an auditor per SLO and to divide the load. The framework implementation also shows a linear scalability of the processing time and memory usage with respect to the number of Facts to be audited.

AURIC framework's functionality is *highly reusable*, which is achieved through the functional decomposition of an audit task into a sequence of subtasks to allow for a modular specification, and through the separation of common audit functionality from SLO-specific auditing logic, as well as a formal language *Sapta* to specify complex audit tasks in full detail. The framework implements the required common audit functionality and offers an API to implement the application logic for auditing a specific SLO. Using AURIC framework, a developer does not need to be concerned about the control of data flow, management of audit data, and data transport. Therefore, the efforts to develop an auditing application based on AURIC framework are largely reduced.

Acknowledgments

The work has been performed partially in the framework of the EU IST Project Akogrimo (IST-2004-004293), the EU IST Project Daidalos II (IST-2005-026943), and the EU IST Network of Excellence EMANICS (IST-NoE-026854).

References

1. Agilent Technologies: *Measuring Web Quality of Service with the New HTTP Test in Fire-Hunter 4.0*; White Paper, Agilent Technologies Inc., November 2002.
2. P. Calhoun, J. Loughney, E. Guttman, G. Zorn, J. Arkko: *Diameter Base Protocol*; IETF, RFC 3588, September 2003.
3. C. de Laat, G. Gross, L. Gommans, J. Vollbrecht, D. Spence: *Generic AAA Architecture*; IETF, RFC 2903, August 2000.
4. S. Duddi: *Demystifying Footprint*; mozilla.org, March 2002.
5. V. I. Fajardo: *Open Diameter Software Architecture*; 2004.
6. G-NE GmbH: *Konzeptionsansatz: Qualitätssicherung in IT-Outsourcing-Projekten mittels einer unabhängigen Prüfinstanz*; Confidential Document, 2002.
7. Hasan: *A Generic Auditing Framework for Compliance Verification of Internet Service Level Agreements*; ETH Zürich, Switzerland, PhD Thesis, Shaker Verlag GmbH, Aachen, 2007.
8. Hasan (Editor): *A4C Framework Design Specification*; Deliverable D341, Sixth European Union Framework Programme, IST Project “Daidalos”, September 2004.
9. Hasan, B. Stiller: *A Generic Model and Architecture for Automated Auditing*; 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM’05), Barcelona, Spain, October 24-26, 2005.
10. Hasan, B. Stiller: *Auditing Architecture for SLA Violation Detection in QoS-Supporting Mobile Internet*; IST Mobile and Wireless Comm. Summit 2003, Aveiro, Portugal, June 2003.
11. Hasan, B. Stiller: *Non-repudiation of Consumption of Mobile Internet Services with Privacy Support*; IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob’05), Montreal, Canada, 2005.
12. Itellix Software: *Wisiba*; Datasheet, 2003.
13. A. Keller, H. Ludwig: *The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services*; Journal of Network and Systems Management, Vol. 11, Issue 1, pp. 57-81, March 2003.
14. H. Ludwig, A. Dan, R. Kearney: *Cremona: An Architecture and Library for Creation and Monitoring of WS-Agreements*; International Conference on Service Oriented Computing, New York, USA, November 2004.
15. J. Skene, W. Emmerich: *Generating a Contract Checker for an SLA Language*; EDOC 2004 Workshop on Contract Architectures and Languages, Monterey, California, 2004.
16. J. Skene, D. Davide Lamanna, W. Emmerich: *Precise Service Level Agreements*; 26th International Conference on Software Engineering, Edinburgh, UK, May 2004.
17. Telemanagement Forum: *SLA Management Handbook*; V1.5. GB917, 2001.
18. D. C. Verma: *Service Level Agreements on IP Networks*; Proceedings of the IEEE, Vol. 92, Issue 9, September 2004.