Journal of Network and Computer Applications 80 (2017) 22–34

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Kerberos: A real-time fraud detection system for IMS-enabled VoIP networks

L. Manunza[a], S. Marseglia[b], S.P. Romano[b,*]

[a] Tiscali Italia S.p.A., Italy
[b] Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione (DIETI), University of Napoli Federico II, Italy

ARTICLE INFO

Keywords:
Fraud detection
VoIP
IMS networks
Complex event processing
Scalability

ABSTRACT

In this paper we present the design, implementation and experimental evaluation of Kerberos, an architecture for the detection of frauds in current generation Voice over IP (VoIP) networks. Kerberos is fed by an On-line Charging System (OCS) generating events associated with the setup, evolution and tear-down of end-user calls in a VoIP network compliant with the IP Multimedia Subsystem (IMS) specification. Such events are properly correlated in order to identify, in real-time, patterns associated with a fraudulent utilization of the Operator's resources. The detection phase can in turn trigger the subsequent remediation actions. Communication between the OCS and Kerberos is based on an asynchronous paradigm, whereas event correlation and analysis are effectively realized through a Complex Event Processing approach. The paper will shed light on both the design and the implementation of the system, whose performance is then evaluated by relying on a real-world dataset of Call Detail Record (CDR) events provided by Tiscali, a well known Italian Operator.

1. Introduction

Nowadays, most telephone operators rely on Online Charging Systems (OCS) to monitor the phone calls made by their users through the generation of events carrying updated information about network resource utilization. Such information is typically used for billing purposes, but it can also be leveraged in order to detect frauds, i.e., identify those users who make a fraudulent (or at least unauthorized) use of the offered service. Fraud detection can indeed rely on two alternative approaches:

- Offline fraud detection: this is the classical approach based on the off-line analysis of the so-called *Call Detail Records* (CDRs). A CDR is a piece of information associated with a call; it is created at the end of the call and contains aggregated data about it (caller, callee, call duration, etc.). With the offline approach, frauds can only be detected after the call has terminated;
- Online fraud detection: this is a more recent approach, based on the dynamic, real-time analysis of call information. With this approach, frauds are detected through the direct analysis of OCS events rather than through off-line CDR elaboration. While more demanding in terms of computational resources, on-line detection allows the operator to spot frauds 'while' a call is still in place and thus take proper actions (e.g., force the call to hang-up, divert the call towards a dedicated fraud management component, etc.) as soon as the potential issue is identified.

In this paper we present *Kerberos*, a dynamically configurable system which is capable to perform online fraud detection while at the same time producing standard CDRs by properly aggregating events generated by the OCS and associated with a single call. To achieve this goal, Kerberos makes use of Complex Event Processing (CEP) techniques to identify, within the real-time flow of events produced by the OCS, subsets (or sequences) of events indicating the presence of a potential fraud. Kerberos has been developed as a high performance processing engine which also exposes a friendly web-based management interface. It has been designed and implemented based on specifications provided by Tiscali, a well-known Italian OTT (Over The Top) service provider that is currently offering, through the *Indoona* platform, unified access (web, mobile, PSTN, etc.) to its cloud-based IMS (IP Multimedia Subsystem) infrastructure. Tiscali has indeed provided us with the overall guidelines for the system, as well as detailed information about the basic fraud templates to be taken into account and a rich dataset containing real-world OCS events collected on-the-field.

The paper is organized as follows. In Section 2 we will present the current state of the art in the field of fraud detection in VoIP networks, with an eye on the most interesting proposals related to real-time

* Corresponding author.
E-mail address: spromano@unina.it (S.P. Romano).

http://dx.doi.org/10.1016/j.jnca.2016.12.018
Received 30 May 2016; Received in revised form 4 October 2016; Accepted 6 December 2016
Available online 09 December 2016

analysis. Section 3 presents an overview of the Kerberos architecture, in terms of building blocks and protocol interactions. System requirements are discussed in Section 4. The main details associated with the implementation of Kerberos are provided in Section 5, whereas Section 6 deals with performance, with a special focus on scalability. Finally, Section 7 summarizes the main results of our current efforts and indicates the most interesting directions of our future work.

2. Fraud detection in VoIP networks: state of the art

A recent survey on the general topic of Fraud Detection can be found in Abdallah et al. (2016).

In this paper we somehow narrow the scope of interest to those researches that have recently focused on detection of frauds associated with the malicious utilization of a VoIP network. Some authors have indeed provided contributions associated with the definition of proper architectural frameworks. Others have instead focused on finding the most effective way to describe (and classify) the numerous types of frauds that a detection system has to deal with during its operation.

The authors of Kapourniotis et al. (2011) present a fraud detection framework for VoIP networks focusing on user profiling. They basically rely on an ontology-driven unsupervised algorithm which allows to derive information about user's behavior starting from an analysis of CDR data. By leveraging a Bayesian Belief Network they detect suspicious behaviors in the stored CDR data and trigger alarms when needed. The authors remark that their proposed approach can be used to generate particular user signatures after parsing an already available CDR data set. Such signatures might then feed an on-line algorithm in order to detect frauds in real-time. In our previous work Chiappetta et al. (2013), we also focused on the extrapolation of user profiles from stored CDR data, with special reference to the identification of a specific category of malicious users, the so-called *telemarketers*. One further CDR-based security architecture, called SAD (SIP Anomaly Detection), can be found in De Lutiis and Lombardo (2009). SAD has been designed as an anomaly detection tool able to analyze Call Detail Records from the security perspective, with a minimal impact on the operator's IMS infrastructure.

Authors of the work in Aziz et al. (2014) propose a distributed monitoring architecture for the detection of attacks to SIP networks. In their paper, they use 'toll fraud' as an outstanding example of these kind of threats. In the authors' interpretation, a toll fraud attack takes place whenever a malicious user generates costs by misusing the extension of another user. This requires that the attacker preliminary succeeds in hijacking a valid extension and then uses it to make calls. In our work, we use the term in a somewhat different context, since we mainly focus on frauds associated with a malicious use of a non-hijacked account, as it will come out from the considerations we make in the following sections.

Along the same lines, an integrated approach to fraud detection and prevention in VoIP networks has been proposed in Hoffstadt et al. (2014). The SUNSHINE architecture is built as a modular composition of prevention and detection components, both at the network and at the application level. It actually leverages firewalling and intrusion detection on top of a distributed sensing system. Furthermore, it relies on CDR processing by properly mixing statistical analysis and artificial intelligence. Finally, SUNSHINE is capable of correlating and aggregating alarms and exploits real-time black-listing based on DNS as a remediation strategy.

An interesting taxonomy of existing VoIP security threats, with special reference to toll frauds, can be found in Gruber et al. (2013). Such a taxonomy is indeed derived from a detailed analysis of data captured on-the-field thanks to the deployment of an ad hoc configured 'honeynet' architecture. The results of the experiments conducted by the authors clearly show how a common pattern that can be found in most modern toll fraud attacks concerns the adoption of techniques allowing the attackers to take advantage from the malicious use of the provider's infrastructure without incurring in any additional costs.

We conclude this section by mentioning the work contained in Rozsnyai et al. (2007), whose ideas, though not directly related to the VoIP ecosystem, show a lot of similarities with the approach we propose for Kerberos. Indeed, the authors of the cited work propose an event-based architecture for the prevention and detection of frauds, by focusing on a demonstration scenario associated with on-line gambling. The proposed architecture, called SARI (Sense And Respond Infrastructure), leverages the same Complex Event Processing (CEP) approach we propose for Kerberos. It is capable to process large amounts of events and provides functions to monitor, configure and optimize business processes in real-time.

3. The kerberos architecture

A high-level view of the Kerberos architecture is sketched in Fig. 1. On the top left part of the picture we have the OCS, which is a producer of charging events associated with ongoing calls. Such events are published onto a dedicated event channel, which the Kerberos engine (on the right side of the picture) is subscribed to as a consumer entity. All of the events eventually pass through the Kerberos rule-based processing engine (right part of the picture), which accomplishes two main tasks: (i) generating (and storing into a dedicated database) an alert whenever a sequence of events matches a specific rule template; (ii) producing and storing in a dedicated database, at the end of each monitored call, a Call Detail Record (CDR) suitable for off-line analysis. The lower left part of the picture shows how Kerberos interacts with a properly authenticated administrative entity by notifying it about alert situations through different channels, namely a push-based web interface, an e-mail message or a phone text message (SMS). As it will be better explained in the following sections of the paper, the administrative entity can indeed interact with the Kerberos back-end engine through a web-based GUI allowing her to perform the following tasks:

- insert new rules into the Kerberos event correlation engine by properly setting the values of a well defined set of parameterized rule templates;
- build new rules which do not match any of the pre-defined rule templates;
- configure the desired notification means associated with the alerts generated by a specific rule.

As it comes out from the high-level sketch described above, interaction between Tiscali's OCS and Kerberos has been devised at the outset as a standard Event Driven Architecture (EDA (McGovern et al., 2006), see Fig. 2), i.e., a framework specifically oriented towards the *production*, *detection*, *consumption* and *reaction* to 'events', that is to say records of a generic activity of interest within the system under

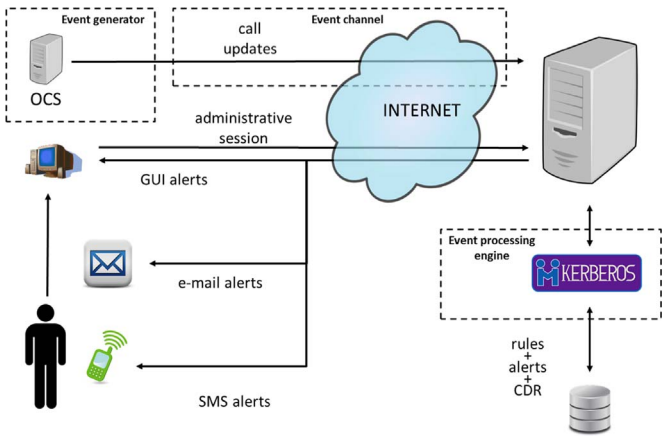


Fig. 1. Kerberos system architecture.

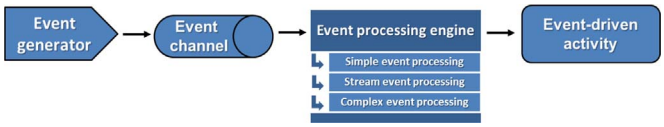


Fig. 2. Event Driven Architecture.

study.

An EDA is typically realized as a loosely coupled distributed system whose components interoperate through the exchanging of messages. In order to cope with the principle of separation of concerns, a critical role in such systems is played by the event channel, which has to guarantee that communication among components happens on the basis of an asynchronous paradigm. This is the reason why event channels are usually implemented through a dedicated *message oriented middleware* allowing for the creation of message queues that operate on the basis of a publish/subscribe mechanism.

Coming to the *event processing engine*, Kerberos embraces an approach falling into the so-called *Complex Event Processing* (CEP) category (Luckham, 2008). With this approach it is possible to create *complex events* starting from a set of *simple events* belonging to one or more *event streams*. This process can be iterated in order to create "event hierarchies" allowing to observe and describe system activities at different levels of detail. At the lowest layer we find simple events corresponding to activities that are actually performed by the system; at the higher layers, on the other hand, we deal with events that have been derived from a set of simple events by leveraging the following categories of event relationships: (i) *temporal*: event A happens before event B; (ii) *causal*: event A causes event B; (iii) *aggregation-based*: event A is an aggregation of events B_1, B_2, …, B_n. The above mentioned relationships can be defined through ad hoc schemes known as *event patterns*, which are specified in a suitable language made available by the event processing engine. When the engine detects the occurrence of a specified pattern among the event streams it is receiving in input, it can undertake a well defined action. In the literature, the information set represented by an event pattern together with its related actions is typically known as a *rule*. Kerberos allows the administrator to define rules associated with the detection of specific patterns within the stream of events represented by Tiscali's OCS updates.

4. Tiscali's fraud detection functions and rule templates

Before delving into the details of how Kerberos has been implemented, in this section we focus on the process that brought us to the definition of a well-defined set of fraud detection rule templates associated with Tiscali's Indoona[1] service. In a nutshell, Indoona is an integrated application allowing users to socially interact (even in groups) through the exchanging of instant messages, as well as by calling and video-calling from either a smartphone or a personal computer. The interesting thing about this application is that it represents a real-world example of how a WebRTC-enabled peer can seamlessly get connected (through an ad hoc gateway entity) to a standard IMS (IP Multimedia Subsystem) architecture. This allows, among other things, a mobile phone or a browser to place a call to either a landline or a different mobile device by leveraging the Internet infrastructure, hence significantly capping costs. Indeed, Tiscali also offers a promotional service allowing 100 min of free calls, each month, when calling landlines in Europe and landlines and mobiles in USA, Canada and China. This promotional service, while appealing for Indoona subscribers, has been a substantial hassle for Tiscali, due to the fact that a number of fraudulent ways of exploiting it have been found by malicious users.

The most challenging swindle impacting Tiscali's infrastructure was indeed an age-old fraud that has found new life now that most corporate phone lines run over the Internet. The scheme behind such frauds is unexpectedly simple. Hackers sign up to lease premium-rate phone numbers (often used for either sexual-chat, or online psychic services, or similar activities skirting laws up to the edge of illegality), from one out of a number of web-based services[2] that charge dialers with high per-minute rates, while giving the lessee a cut of the revenues. In the United States, such premium-rate numbers are easily identified by "1-900" prefixes, and the law imposes that callers are informed they will be charged higher rates. But in other countries, like, e.g., in Croatia, Poland and Estonia, they can be much trickier to spot (and can thus easily blend with standard, non-premium numbers). The payout to the lessees can be as high as 0,25€per minute spent on the phone. What typically happens, with this kind of frauds, is that hackers break into a company's (or even a private user's) phone system and make calls through it to their premium number, typically during night or over weekends, when it is harder (for the unfortunate victim) to figure out that something strange is going on. With high-speed computers, they can place a number of calls simultaneously, hence forwarding a huge quantity of phone calls to the pay line. The final goal is for the hacker to get a share of the charges.

With Indoona promotional offer of 100 min of free calls, the above mentioned fraud becomes even easier, since it does not require at all that the hacker succeeds in hijacking someone else's phone number before being able to place the calls. The only thing that is needed is for the hacker to properly register as an Indoona user. The registration process actually requires that a phone number is provided and that such a number is used by the system to send back to the registrant a verification code via SMS. Unfortunately, this kind of check over the registrant, which is based on the assumption that all registered Indoona profiles are associated with SIM cards, can be easily circumvented by relying, once again, on on-line services[3] which basically allow users to receive SMS text messages in the absence of a real SIM card. By leveraging the above mentioned means, a malicious user can easily collect a relevant number of 'fake' Indoona profiles by running a script which automates the registration procedure.

From the discussion above, it follows that a good fraud detection system for Tiscali needs to be capable of keeping track of the most active users of the architecture, in terms of both 'callers' (users who place calls at the highest frequency) and 'callees' (users who receive calls at the highest frequency). Such users are indeed clear indicators of ongoing out-of-profile activities, among which there is the highest probability of finding malicious behaviors. The 'callers' perspective is useful to spot malicious users placing automated calls, whereas the 'callees' perspective helps identify the targets of such calls (hence telling premium-rate numbers apart from standard ones).

A further detection parameter is definitely represented by the capability of leveraging the 'free calls' promotional service offered by Indoona. A good detection system will hence need to keep track of those users who generate a significant amount of 'free' traffic.

As a matter of fact, smart malicious users have also found ways of exploiting non-free calls when implementing their attack plans. The idea behind these specific attack patterns is to make charged calls to premium-rate numbers: the cost of the call (incurred by the hacker) is paid back with the called premium-rate number revenues, which are typically at least an order of magnitude higher. This consideration entails that the fraud detection system cannot just rely on the analysis of free calls, but rather has to also keep track of those users who generate a significant amount of paid traffic.

In summary, the envisaged detection system must compute real-time statistics about managed calls, generate an alert (e.g., via e-mail and/or SMS) as soon as a suspicious behavior is detected and be

[1] http://www.indoona.com

[2] See, e.g., http://www.premiumrateinternational.com/

[3] See, e.g., http://www.it.receive-sms-online.info/

flexible enough to allow for the definition of a customizable set of rules for the identification of users' suspicious behaviors. It is well-known, in fact, that attack patterns keep on changing over time, hence calling for an extreme flexibility on the opponent detection systems.

Kerberos has hence been conceived at the outset as a customizable architecture allowing its administrators to easily add, update and delete fraud detection rules, while also offering a basic set of 'blueprints' under the form of parameterized rule templates. The templates in question are built around a number of different facets of the system and call for the capability of tracking both the evolution of a single call and of a number of different calls sharing one or more property and concurring to the definition of a potential fraud pattern.

Based on the general considerations about the typical frauds perpetrated over Indoona, we have identified the following non-exhaustive list of parameterized templates:

1. A PSTN (Public Switched Telephone Network) number is called for free by at least [X] different Indoona users in the past [Y] hours;
2. A PSTN number receives at least [X] minutes of free calls in the past [Y] hours;
3. A specific Indoona caller consumes more than [X] minutes of free calls towards just [Y] PSTN numbers in the past [Z] hours;
4. A specific Indoona caller consumes more than [X] €of non-free calls in the past [Y] hours.

The mentioned blueprints try and capture most of the information that is needed in order to reliably spot fraud attempts like the ones we discussed at the beginning of this section. Most of them are indeed targeted at identifying all the potential situations in which someone is making a significant number of calls (either for free or at a very low charge) towards a PSTN number behind which there are chances that a premium-rate service is lurking.

Clearly, these are the templates that we devised for the specific case of the Indoona architecture managed by Tiscali. Nonetheless, the system we architected can be easily tailored to different operational scenarios, since it allows for the introduction (even "on-the-fly", i.e., at run time) of newly created rules defined by the system's administrator. The implementation of such user-defined rule templates is briefly discussed in Section 5.8.1.

4.1. OCS events format

Since OCS events play a major role in the Kerberos architecture, we are going to take a closer look at them. Each such event is indeed a tuple comprising the following pieces of information:

- `session_id`: a unique identifier associated with a call;
- `caller`: extension of the user placing the call;
- `callee`: extension of the user receiving the call;
- `dest_domain`: terminating domain for the call;
- `term_cause`: a standard identifier for the call termination cause, in full compliance with ITU-T Q.850 Recommendation (Uberti and Jennings, 1998);
- `start_time`: time at which the call has started;
- `used_balance`: credit consumed by the caller;
- `used_time`: call duration;
- `req_type`: indicates whether the tuple in question refers to the initiation of a new call (req_type=0), to an update associated with an ongoing call (req_type=1), or to the termination of an existing call (req_type=2);
- `timestamp`: the exact time at which the tuple in question has been created at the OCS.

In summary, the OCS generates, for each call, exactly one event with *req_type=0*, zero or more events with *req_type=1* and exactly one event with *req_type=2* upon termination of the call. Some of the fields in the generated tuples (*session_id*, *caller*, *callee*, *dest_domain*, *start_time*) never change during a call, whereas the others can be updated when *req_type* equals either 1 (*used_balance*, *used_time*) or 2 (*used_balance*, *used_time* and *term_cause*).

5. Kerberos implementation

In this section we focus on the implementation of Kerberos. As already anticipated, Kerberos has been structured as a COMET-based (Crane and McCarthy, 2009) *Rich Internet Application* (*RIA*) offering to the administrator a friendly Web GUI that can be used both to configure the overall behavior of the system and to graphically present push-based notifications (call statistics, alerts, etc.) coming from the core system components running in the back-end. Apart from the real-time functionality which represents the main focus of this paper, Kerberos also offers a number of off-line CDR analysis features by seamlessly integrating with a widely deployed open source tool called CDR-Stats.[4] With respect to this last point, the CDRs generated by Kerberos (as soon as a call terminates, i.e., upon reception of a tuple with *req_type=2*) get stored in a database that feeds an instance of the CDR-Stats system running as one further backend component and whose web interface is integrated in the Kerberos Web GUI as an independent tab.

5.1. Kerberos event engine

Coming back to the real-time functionality of the system, the core component is definitely represented by the Event Engine which is in charge of performing the mentioned Complex Event Processing tasks. For this part of the system Kerberos relies on *Esper*[5], an open source CEP engine which can be easily integrated into Java-based applications.

5.1.1. Stateless vs stateful processing in esper

The Esper engine leveraged by Kerberos is capable of going beyond stateless processing of "streams", where by the term stream we refer to a flow of messages (often called events) spread over a well-defined time-frame. Indeed, when analyzing a stream, the needed processing might be either stateless or stateful, depending on the required use of working memory. More precisely, working memory can be associated with the slice of memory used by the processing engine to make inferences, processing or computing. Actions may or may not change the state of the working memory. For example, if a stock's prize change is an event and we also store the number of occurrences of the prize reaching a threshold value of 'X′ $ during the day and use this 'fact' (number of threshold breaches) to take other action, the processing would be called stateful. If we do not take into account the number of breaches during the day and treat every event (stock's prize reaching X $) independently to take an uncorrelated action (like, e.g., sending a broadcast notification), the processing would be called stateless. Stateless processing does not really call for a Complex Event Processor (CEP) engine and could instead rely on simpler systems called 'rule engines'. Rules define actions for a given condition. For example: "If the stock's prize reaches a value of X, send a broadcast notification to subscribers'. Stock's prize reaching X is an event. Sending out the broadcast notification is an action corresponding to the event. The action would happen when the condition is met. An engine that is able to process rules such as above is a rule engine.

It is typical to use the term Rule Engine with stateless processing and CEP with stateful processing of events. As already stated, the "C" in the CEP stands for Complex, which generally means that the engine takes into account multiple events and often the state of facts or

[4] http://www.cdr-stats.org
[5] http://www.espertech.com/

concepts (for example the number of threshold breaches) to take an action.

CEP is often used for pattern detection. Let us understand this better by using the stock's prize example we have been using above. “Given the prize has reached a value of X (Event 1) and given the number of such occurrences is 7 or more (facts), send an alert to the financial control department”.

CEP is also often used for sliding window implementation, which uses time interval as a condition. An example could be a flight sending information to the central monitoring system. If a flight sends out GPS information at specific points in time and has not sent the any GPS update during the last 30 min, the central monitoring system sends an alert to the closest air traffic control tower for the last received GPS location. As it comes out from the example above, in this case the ‘time’ is itself an event and the passage of time (i.e., a (sliding window) is a condition.

We finally remark that CEP is also used for forward and backward chaining of events. To summarise Rule engines are good for stateless processing (any preprocessing, simple rule execution) and CEP for stateful processing (correlate or aggregate events over a time window, pattern detection involving multiple events or facts or in general anything that requires the engine to be aware of the state of the working memory to take an action). Esper is actually suitable for both stateless and stateful processing needs. As it will come out from the following discussion, the events we need to deal with for fraud detection purposes fall, in the most general case, in the realm of stateful processing.

5.1.2. Esper inner workings

With Esper, events can be associated with Java objects (the engine actually uses the term *POJO*, standing for Plain Old Java Object). In a nutshell, an event is represented as a Java class with a member variable for each event property and *getter* methods adhering to the Java beans convention. With this approach, as soon as an event occurs, an application which uses Esper simply creates an object belonging to the class that has been associated with that specific event. In order to define the criteria for the detection of the occurrence of either a specific event or a scheme representing a well defined relationship among events, Esper makes use of a declarative language called EPL (*Event Processing Language*), whose syntax, while recalling standard SQL, actually differs from it since it works on event series rather than data tables. An EPL expression (also called *statement*) hence allows to infer (or aggregate) information starting from one or more event streams.

Esper adopts a publish/subscribe approach: Epser's engine sends information derived from a statement to ad hoc defined *listener* objects that have previously performed an explicit *subscribe* operation. In practice, a listener has to obtain a reference to an instance of the Esper engine and use it to both register a specific statement and subscribe itself as a listener of the events that such a statement is going to generate.

Kerberos also leverages some advanced features made available by Esper, among which we cite the so-called *view*. A view is simply a subset of the events that have to be analyzed by the event processing engine. A couple of standard views, called *windows*, are available in Esper: (i) *length window*: tells the statement to only process the latest *n* events in a stream; (ii) *time window*: tells the statement to only process the events falling in a well defined time frame. Windows are often employed in conjunction with a number of predefined *aggregation functions* (e.g., *count(property)*, *sum(property)*, etc.) which take an event property as an input parameter and, based on the values that such a property assumes within the events belonging to the window in question, compute a single value by applying a well-known function (count, sum, average, etc.). As an example of the application of the above mentioned concepts in Kerberos, we report below the Esper statement implementing rule template number 2 associated with REQ-9 of Section 4:

```
SELECT callee, sum(freeTime) FROM ClosedCallEvent.
win:time(“+timeRange+”) WHERE destDomain LIKE ‘%PSTN%’
GROUP BY callee”.
```

As the reader will easily recognize, the above statement allows Kerberos to keep track of the duration of free calls received by a specific PSTN user in a well defined time frame.

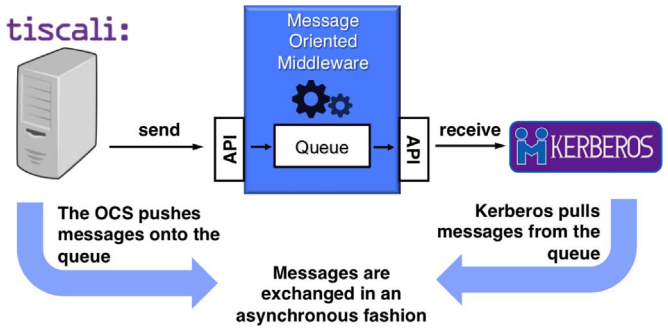


Fig. 3. The role of RabbitMQ as a Message Oriented Middleware.

5.2. Event channel

With reference to the Event Driven Architecture discussed in Section 3, the event channel between Tiscali's OCS and Kerberos has been implemented through the *RabbitMQ*[6] framework. RabbitMQ is an open source software allowing for the exchanging of messages among heterogeneous applications operating in a distributed environment. In this respect, such a framework belongs to the so-called *Message Oriented Middleware* (*MOM*) category, since it enables asynchronous communication among loosely coupled interacting components (see Fig. 3). A MOM is exactly what was needed in our case, since we had loosely coupled interaction among our system requirements.

OCS events get published onto the event channel in JSON (JavaScript Object Notation) format. Kerberos, acting as the RabitMQ consumer, parses the events it receives and creates the proper Esper event objects, as described above.

5.3. The observer pattern in kerberos

In order to clearly separate event processing from event visualization activities, in Kerberos we made use of the well-known *Observer* pattern.

Such a pattern envisages the presence of: (i) a *Subject* object whose state can change over time; (ii) zero or more *Observer* objects which are interested in receiving state change notifications associated with the *Subject* object.

The Web application we devised for Kerberos is built around the classical *Model-View-Controller* (*MVC*) approach. It contains some real-time graphs about ongoing calls and alerts, fed with data generated by a number of Kerberos inner classes playing the role of the *Subject* in the Observer pattern. The class acting as a controller in the MVC architecture of the system (and which looks after graph updates in the Web consolle) is hence playing the *Observer* role with respect to the mentioned Subject classes.

5.4. Event representation and management

The first part of the sequence diagram in Fig. 4 illustrates how Kerberos manages OCS update events. The classes *RabbitMqProducer* and *RabbitMqConsumer* represent, respectively, the OCS-side process producing events and the Kerberos-side process consuming them, as briefly illustrated in Section 5.2.

[6] https://www.rabbitmq.com/

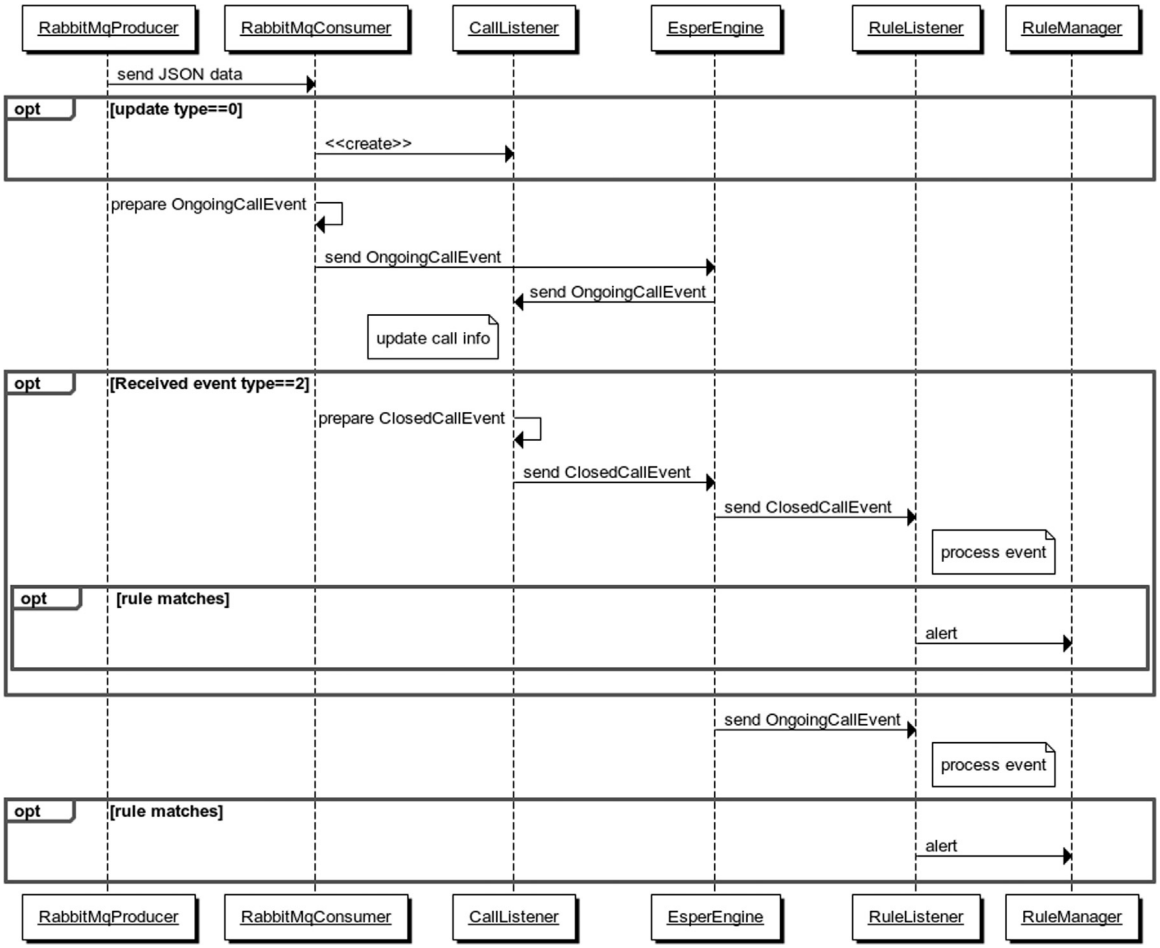


Fig. 4. Sequence diagram associated with the management of events produced by the OCS.

The class *CallListener* represents a single call between two users. It is created upon reception of an OCS event having the *req_type* field equal to 0. Once created, a *CallListener* instance has to subscribe to updates (i.e., events with the same *session_id* field of the tuple which triggered the listener's instantiation, but with $reqtype = 1$) associated with the specific call in question. Each such event is used to properly update call statistics (thanks to fields like *duration*, *used_balance*, *free_time*, etc.). Eventually, a terminating tuple (i.e., an event with the same *session_id* field of the tuple which triggered the listener's instantiation, but with $reqtype = 2$) will arrive, hence triggering the following actions: (i) creation of a standard CDR associated with the call, suitable for logging purposes as well as offline analysis; (ii) generation of a brand new event, called *ClosedCallEvent*, which represents a signal that the call in question is terminated. Such an event can be used by Kerberos during what we call “inter-call processing” activities, i.e., all those event processing tasks that infer information from a set of calls rather than just computing statistical summaries associated with a single call.

5.5. Pre-defined rules implementation

As explained in Section 4, Kerberos is configured to operate with at least a set of four pre-defined rules whose structure has come out from the requirements analysis and system specification phases. The class diagram in Fig. 5 illustrates how such rules have been implemented. In summary, each detection rule properly specializes the behavior of a class called *RuleListener* (which in turns implements the standard *UpdateListener* interface made available by Esper) by customizing the implementation of an abstract method called *createExpression*(), whose goal is the creation of the EQL statemet associated with the rule in question. Each such class creates the related Esper statement and subscribes itself as a listener of the update events associated with it.

The companion class called *RuleManager* is instead used for management purposes and hence takes care of rule creation and deletion, as well as notification procedures (i.e., logging, sending alert information through SMS and/or e-mail to the specified set of recipients, etc).

5.6. Inter-call rules

The middle box in the sequence diagram in Fig. 4 illustrates what happens in the system upon reception of an update event characterized by $reqtype = 2$. As already anticipated, such an event represents a signal that an ongoing call has just terminated. In such a case, the *CallListener* object associated with the call in question completes the computation of call statistics and prepares a new event (called *ClosedCallEvent*), which will be in turn analyzed by what we have called ‘inter-call rules’. The newly created event is sent to the Esper engine and from there notified to the registered rule listener classes introduced in the previous subsection. If the received event matches the specific pattern associated with any such listener, the *RuleManager* is contacted and an alert is raised.

5.7. Intra-call rules

The last part of the sequence diagram in Fig. 4 gives a complete picture of the operation of the system, by showing the behavior of

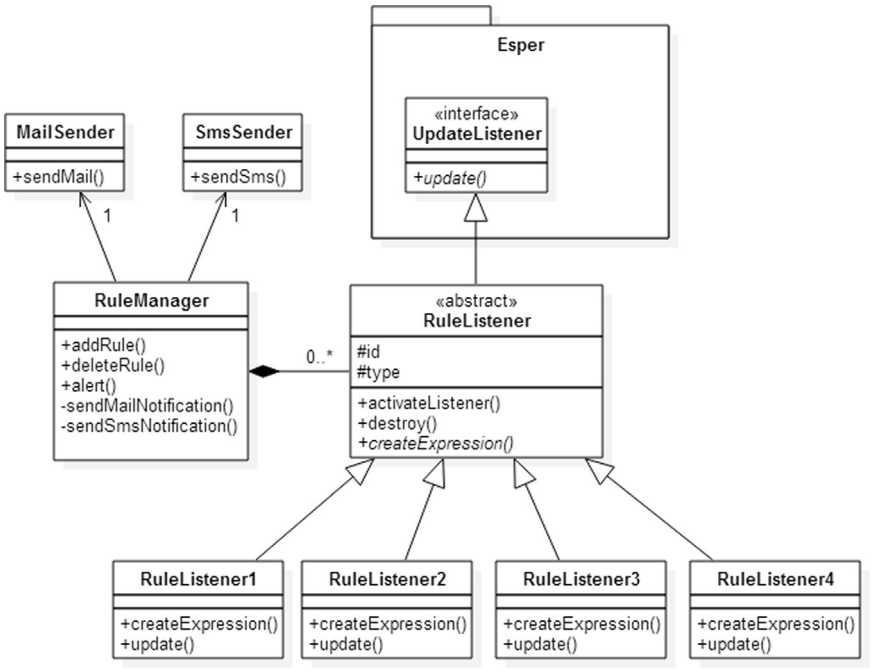


Fig. 5. Class diagram associated with predefined rules.

Kerberos in case of reception of an event associated with an ongoing call (i.e., characterized by *reqtype* = 1). Such events are actually consumed by those classes of the system which implement the so-called 'intra-call' logic. By focusing on the final part of the diagram, we can notice how Esper sends a copy of the *OngoingCallEvent* also to customized versions of the afore-mentioned *RuleListener* abstract class, whose main purpose is the definition of a specific intra-call statement compliant with the generic pattern reported below:

`SELECT <property> FROM OngoingCallEvent [additional clauses]`.

The creation and activation of intra-call rule patterns is illustrated, among other things, in the next subsection.

5.8. User-defined rules creation

Kerberos also allows for the creation of user-defined rule templates. This is achieved through the graphical interface reported in Fig. 6. Such an interface basically allows for the definition of a set of generic rules, each associated with a specific pair 'caller/callee' and a well-defined set of constraints (number of free/paid seconds consumed by the call, cost associated with the call, duration of the call), in a predetermined time frame.

A corner case is represented by the creation of a rule in which either the caller, or the callee (or both) are not specified. In such a case, we do not talk anymore of *a caller* and *a callee*, but rather specify either incoming or outgoing calls between *any* pair of users (see Fig. 7). With respect to the indication of a call's 'direction', we invite the reader to ponder the difference between a rule like "Any user makes calls towards any other user, for a total of at least 100 min" and "Any user receives calls from any other user, for a total of at least 100 min". As Fig. 7 shows, in this case we also add a further filter (in the constraints section) associated with the specification of the prefix of either the caller or the callee (or both). The mentioned constraints can be used, e.g., to spot calls to and from a well-defined geographical area, associated with a specific prefix code.

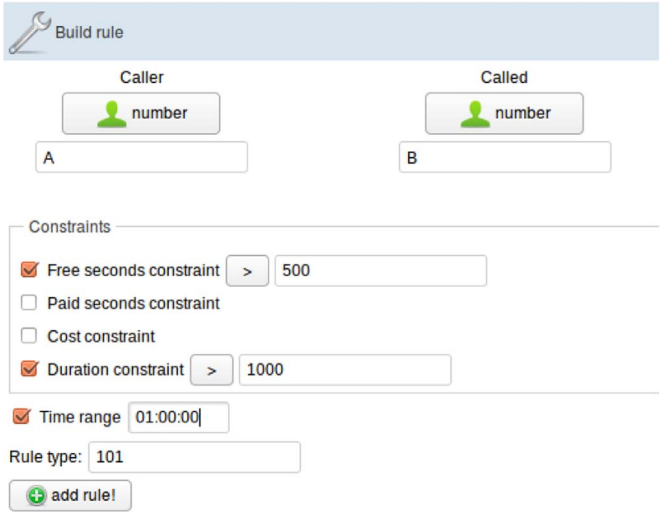


Fig. 6. Web GUI for user-defined rules construction.

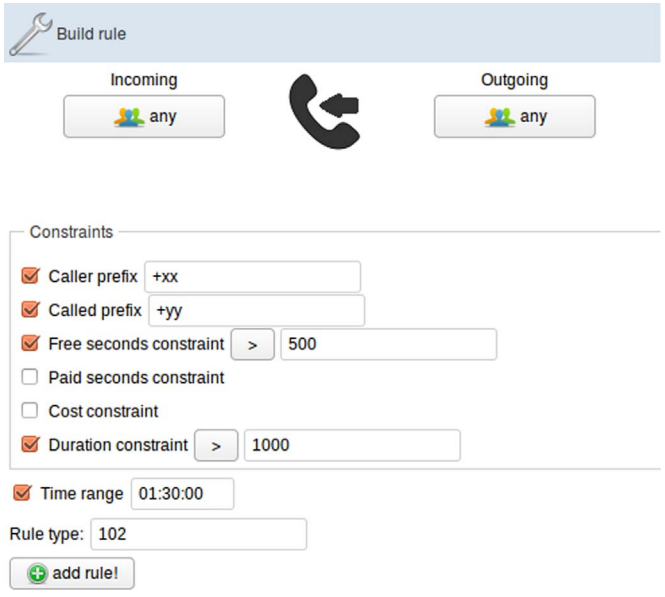


Fig. 7. Building a 'generic' rule.

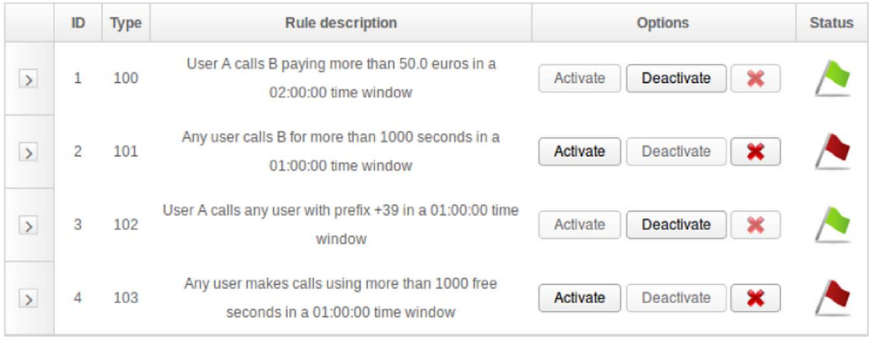


Fig. 8. User-defined rules list example.

Once all of the parameters needed in order to build a new rule have been inserted, the user can add it to the available rule catalog. A newly created rule is not necessarily 'activated' in the system, since the related Esper statement in not yet registered and hence the rule has no impact on the analysis of the events received from the OCS. The activation process is explicitly triggered by pressing the 'activate' button on the system's GUI. Similarly, rules can be either deactivated or deleted from the system (see Fig. 8).

5.8.1. User-defined rules implementation

User-defined rules have been implemented similarly to pre-defined ones: they are associated with Java classes, subscribe as listeners of specific Esper statements and send alerts whenever the pattern they implement is matched. The major difference between the two categories of rules resides in the fact that user-defined rules get created *during* system operation and hence need to be compiled and loaded at run time. To achieve this goal, we leveraged Java code injection, as made available by a well-known library called *Javassist*[7], which enables Java programs to define new classes at runtime, as well as modify a class file when the Java Virtual machine (JVM) loads it. Javassist provides two levels of API: source level and bytecode level. With the former API, users can edit a class file as they are used to do (i.e., with the vocabulary of the Java language) with no need to master Java bytecode specifications. Javassist compiles it on the fly. When developing support for user-defined rules in Kerberos, we opted for the source level API.

6. Performance assessment and system's scalability

Kerberos has been the subject of a thorough experimental campaign aimed at assessing its performance. Before carrying out such a campaign, we have preliminarily performed a number of functional tests whose goal was to verify that the system behaved as expected when fed with generic event patterns. The functional tests, which are omitted here for the sake of brevity, have been based on ad hoc generated event patterns and have proved that: (iii) both inter-call and intra-call rule instances perform as expected; (ii) alerts are correctly generated (and logged), in the presence of both the pre-defined set of rules and a variegated set of user-defined templates; (iii) a new CDR is correctly generated as soon as a call ends.

The technical specification of the server used for the trials is the following: (i) 8 CPU Intel(R) core(TM) i7-4770 s @3.10 GHz; (ii) 16 GB of RAM memory; (iii) Ubuntu 14.10 LTS Operating System.

To generate the test load we have developed a dedicated stressing tool. The tool, written in Java, emulates the behavior of the OCS by acting as the producer of the events published onto the RabbitMQ queue which Kerberos connects to in order to asynchronously receive call-related information. OCS events are actually retrieved from a database allowing us to properly reproduce a well-defined call scenario in order to make comparative evaluations under different configurations of the system. The stressing tool can hence be configured with the following three parameters: (i) the specific database containing OCS events; (ii) the specific database table from which events have to be fetched; (iii) the specific event generation frequency.

In order to let Kerberos work with a realistic set of events, we have made use of a real dataset represented by a copy of Tiscali's event logs associated with a period of time of one week, between December 1^{st} 2014 and December 7^{th} 2014. Such logs contain 562.533 events corresponding to 119.034 calls. For privacy reasons, we're not going to disclose any detailed information about the logs in question. This does not at all undermine the results of the analysis, since what is needed for our purposes is just high-level information about call patterns, generic users' behaviors and capability of the system to keep the pace of OCS event generation as long as the overall load of the system increases.

Fig. 9 reports the profile of the analyzed traffic, together with its slightly increasing trendline (in red).

The above reported data allow us to draw some considerations. First, we can state that a single call is associated, on average, with around 5 (namely, 4,72) events. If we have 119.034 calls in a week, we basically have to consider around 17.000 calls per day, which means 708 calls per hour, or, equivalently, about 11,8 calls per minute. In terms of events, we then have about 55,74 events per minute, i.e., 0,92 events per second. An event generation frequency of 1 event/sec can hence be considered as close as possible to Tiscali's current operational scenario.

6.1. Long-run test

The so-called "long-run" test has been carried out with the following parameter set: (i) event generation frequency=250 msec; (ii) rule instances=8 (2 instances per pre-defined category); (ii) test duration=8 h. Under these conditions Kerberos has processed 150.310 events associated with 32.480 calls related to an actual time frame of about 36 h. During the test, 3.222 alerts have been generated, as reported in the scatter plot in Fig. 10.

CPU and memory utilization data are instead reported in Fig. 11. As expected, the picture clearly shows that Kerberos is a memory-intensive application. CPU utilization is in fact definitely low for the entire duration of the test. Coming to memory, its average utilization level stays around 500 MB with the mentioned (slightly increasing) traffic profile. We can hence claim that Kerberos shows good performance when subjected to a realistic load.

6.2. Stress tests

In this section we focus on the results of a dedicated stress-testing campaign we devised for our system. We will first focus on increasing the event generation frequency (given a fixed set of rules). Hence, we will analyze the behavior of the system in the presence of an increasing

[7] http://jboss-javassist.github.io/javassist/

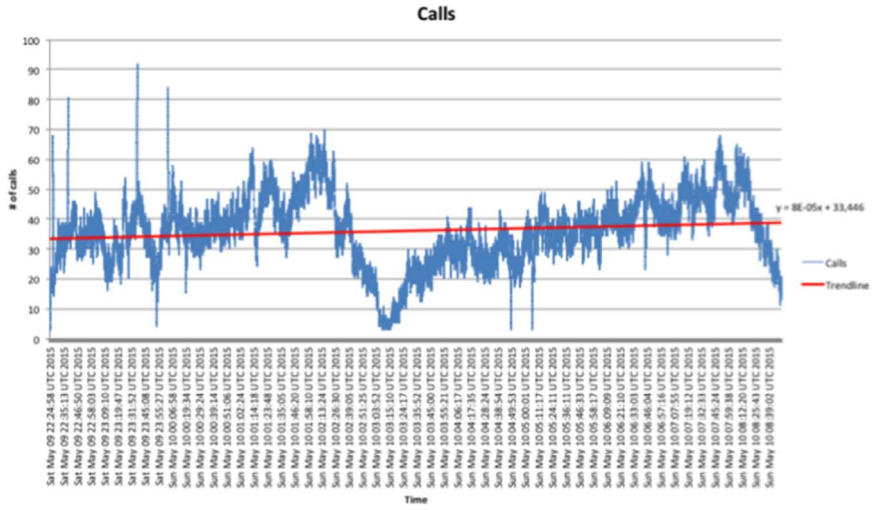


Fig. 9. Tiscali's Indoona traffic profile between December 1st 2014 and December 7th 2014. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

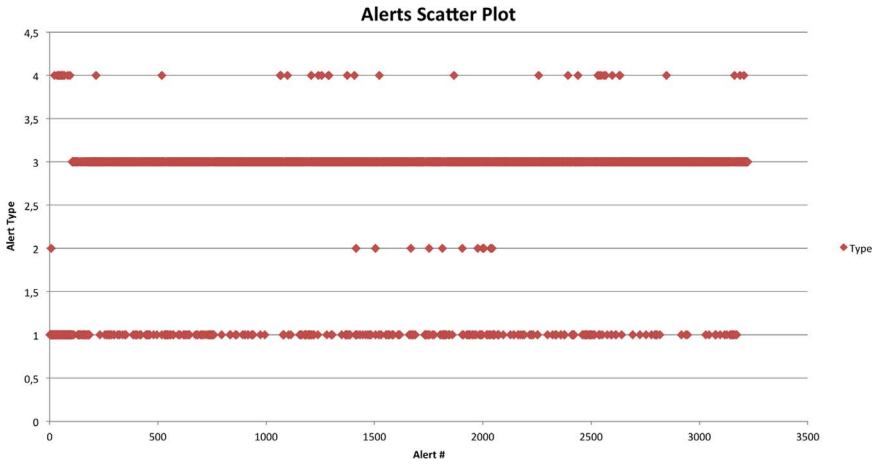


Fig. 10. Alerts generated during the long-run test.

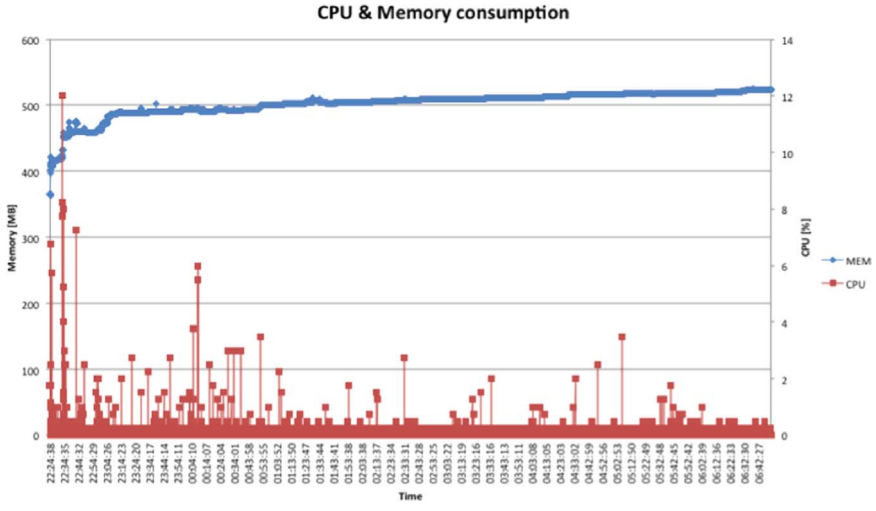


Fig. 11. CPU and memory consumption measured during the long-run test.

number of (either pre-defined or user-defined) rules.

6.2.1. Increasing OCS event generation frequency

In this testing scenario, we progressively increase the rate at which events are sent to a Kerberos system hosting the following set of pre-defined rules:

1. `An Indoona user makes free calls to more than 2 callees in a 1 hour time window`;
2. `An Indoona user makes more than` 6.000 s `of free calls in a`

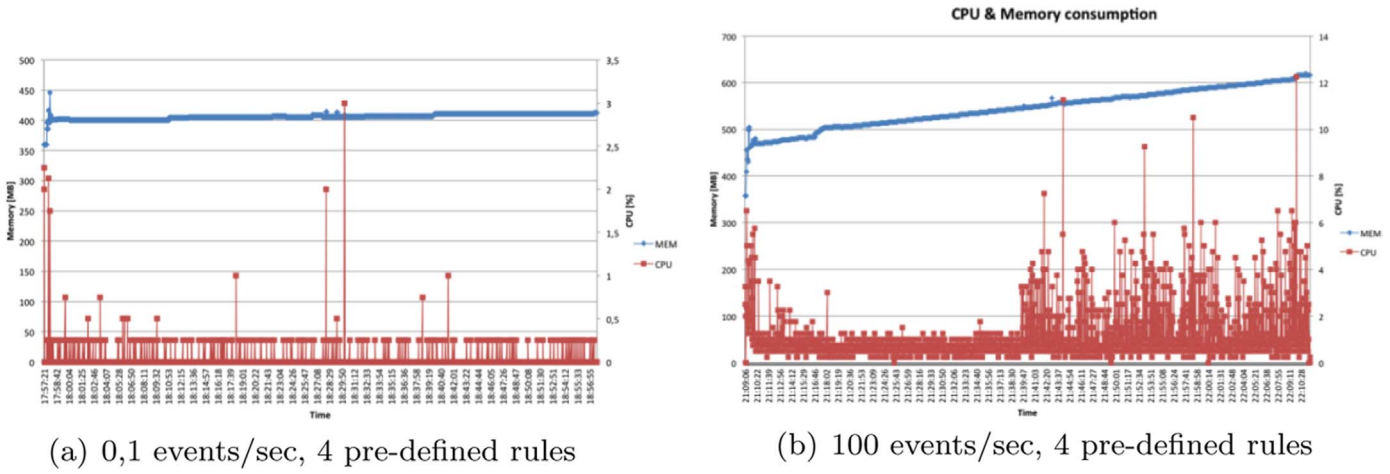


Fig. 12. Stress testing results: increasing event generation frequency. (a) 0,1 events/sec, 4 pre-defined rules, (b) 100 events/sec, 4 pre-defined rules.

`2 hours time window`;

3. `An Indoona user makes more than` 100 s `free calls towards at most 1 callee in a 1 hour time window`;
4. `An Indoona user spends more than` 1€ `on charged calls in a 1 hour time window`.

We have run one-hour-long tests and considered an event generation frequency of, respectively, 0,1 – 1 – 10 and 100 events/sec. For the sake of brevity, we report in Fig. 12 the results associated with the two extreme cases (0,1 and 100 events/sec). The graphs show both memory and CPU consumption. For the picture on the left, Kerberos has analyzed 97 calls and has not generated any alert. For what concerns the picture on the right, the system has instead processed (in just one hour, corresponding to a real-world profile of about 4 days) 79.050 calls, for which 2.887 alerts have been produced. As witnessed by the CPU and memory curves in the graph, the system attains good performance also in this somewhat extreme working condition.

6.2.2. *Increasing rule instances*

In this testing scenario we have fixed the event generation frequency (at a value of 10 events/sec) and progressively increased the number of rules (both pre-defined and user-defined) configured into the system. In the case of pre-defined rules, this has been done by simply replicating the basic rule set we described in the previous subsection. The increase has been done in a stepwise fashion, 5 rules at a time, starting at 5 and stopping at 50 rule instances. The same holds true for the user-defined class, whose basic set has been constructed by choosing 5 templates covering a widely differentiated configuration scenario. Once again, in Fig. 13 we just present the results of the analysis in the most challenging scenarios, namely those related to the presence of 50 rule instances (either pre-defined or user-defined).

The left-hand graph shows CPU and memory consumption for the pre-defined rules case. We remind the reader that the test lasted for about one hour. During this time frame, Kerberos has analyzed 7.768 calls, raising a total of 1.521 alerts. The curves in the graph show how Kerberos is able to easily keep the pace of OCS-generated events in the mentioned setup.

Coming to user-defined rules, the graph on the right reports a slightly higher load of the system, in terms of both CPU and memory consumption. This is actually due to the fact that this specific set of rules has led the system to generate 18.818 alerts (while analyzing 7.818 calls). Such alerts, for which a scatter plot is reported in Fig. 14, are an order of magnitude higher than those associated with the pre-defined rules scenario. Also with this last setup, Kerberos has nonetheless demonstrated its ability to cope with a massive events stream whose features have to be properly analyzed (and put into correlation) in real-time.

6.3. *Profiling kerberos*

As witnessed by the above discussed scalability figures, by leveraging the Esper framework Kerberos has been made capable of handling very high throughput streams with very little latency between event reception and output result posting. Kerberos itself, as well as Esper, runs on top of a JVM, so at least some familiarity with JVM tuning is needed in order to arrive at optimal performance. Key

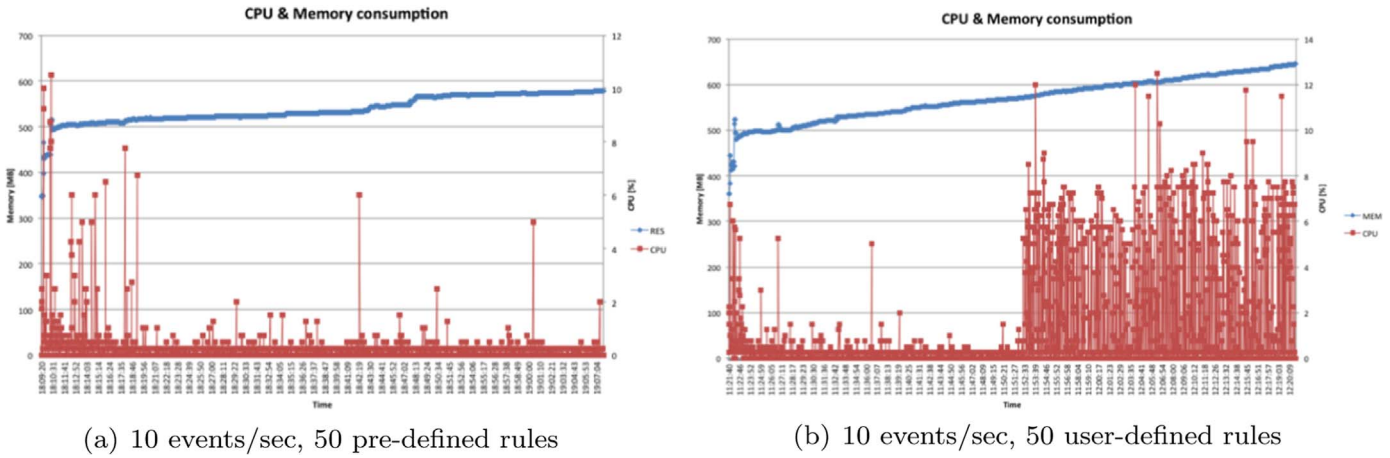


Fig. 13. Stress testing results: increasing the number of rules. (a) 10 events/sec, 50 pre-defined rules, (b) 10 events/sec, 50 user-defined rules.

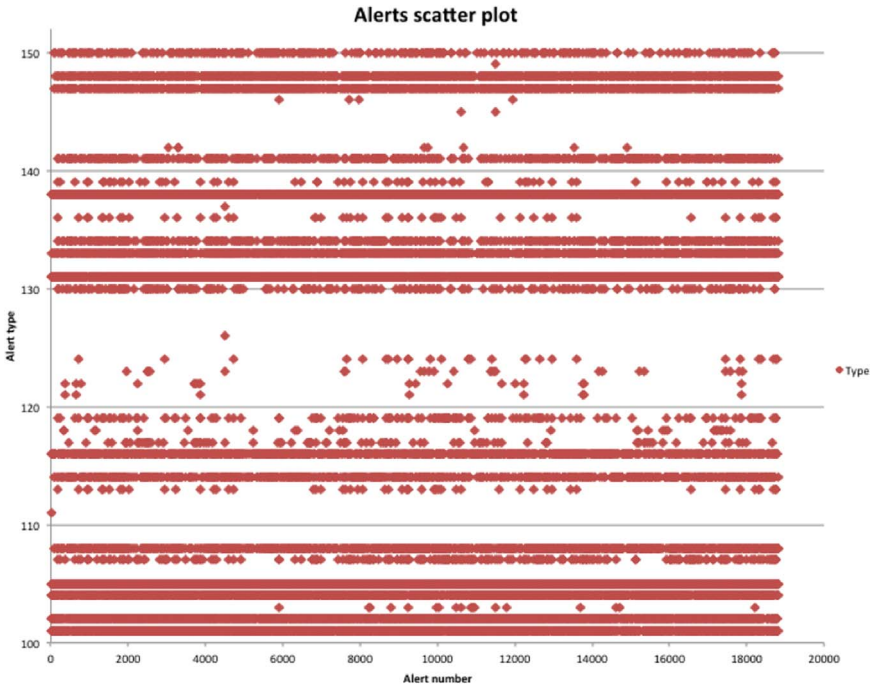
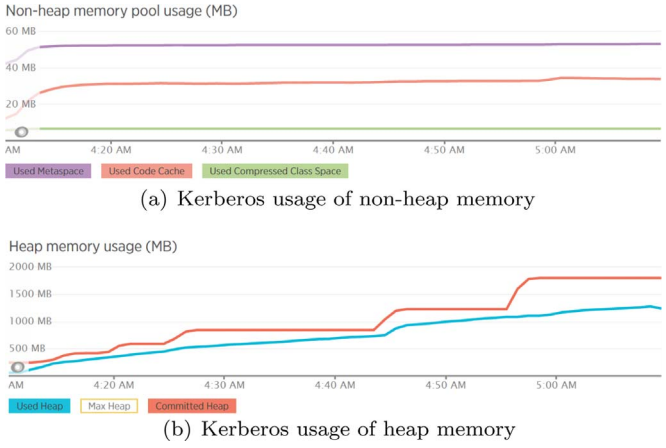


Fig. 14. Alerts scatter plot with 10 events/sec and 50 user-defined rules.

parameters to be taken into account include minimum and maximum heap memory, as well as so called 'nursery' heap sizes. Statements with time-based data windows, in particular, can consume large amounts of memory as their size and length can become large as well. In particular, for time-based data windows, one needs to be aware that the memory consumed depends on the actual event stream input throughput. Event pattern instances also consume memory.

Kerberos management console receives output events from Esper statements via strongly-typed subscriber POJO objects. Such output events are delivered by the application or timer thread(s) that sends an input event into the engine instance. Furthermore, the processing of output events that a listener (or subscriber) performs, temporarily blocks the thread until the processing completes and may thus reduce throughput. We found out that it is therefore highly beneficial to process output events asynchronously and not block the Esper engine while an output event is being elaborated by an application's listener, especially if the listener itself performs blocking I/O operations.

Since we were interested in understanding in more depth the potential performance bottlenecks of Kerberos, we also carried out some fine-grained profiling of the entire application's state, by leveraging the well known *New Relic* Application Performance Monitoring (APM) framework. Based on the consideration that the key performance indicator is in our case represented by dynamic memory allocation and management, we focused on monitoring the behavior of heap memory. This is justified by the fact that Kerberos works, in the most challenging cases, with time-based processing of aggregated event streams. The larger the time window, the higher the load (especially the memory load) on the detection system. Also, rules that require the system to perform a lot of "join-like" operations (by borrowing a term coming from databases and referring to queries which span across multiple tables) on the monitored event streams definitely incur in a higher overhead. As expected, heap memory consumption turned out to be the critical parameter to monitor, as witnessed by the graphs in Fig. 15, related to a typical high-load (i.e., 100 events/sec and 100 active time-based rules) execution of the system. The graphs show that: (i) consumed heap memory is an order of magnitude higher than non-heap memory, on average; (ii) non-heap memory stays constant over time (Fig. 15(a)), whereas heap memory keeps on growing (Fig. 15(b)) as long as the system tracks long-lived event streams.



(a) Kerberos usage of non-heap memory

(b) Kerberos usage of heap memory

Fig. 15. Heap vs Non-heap memory allocation. (a) Kerberos usage of non-heap memory, (b) Kerberos usage of heap memory.

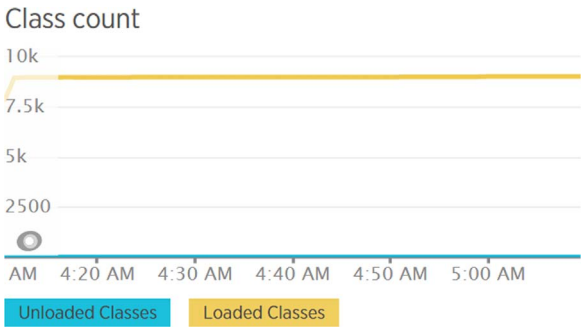


Fig. 16. Kerberos allocated classes during the profiling experiment.

The mentioned memory footprints are associated with the management of an overall number of about 9.000 classes, as showed in Fig. 16.

Delving further into the details of how the heap is actually managed during execution, we herein briefly recall that Java memory space is divided into three regions (or *generations*) for the sake of garbage collection: (i) *New* Generation; (ii) *Old* (or *tenured*) Generation; (iii)

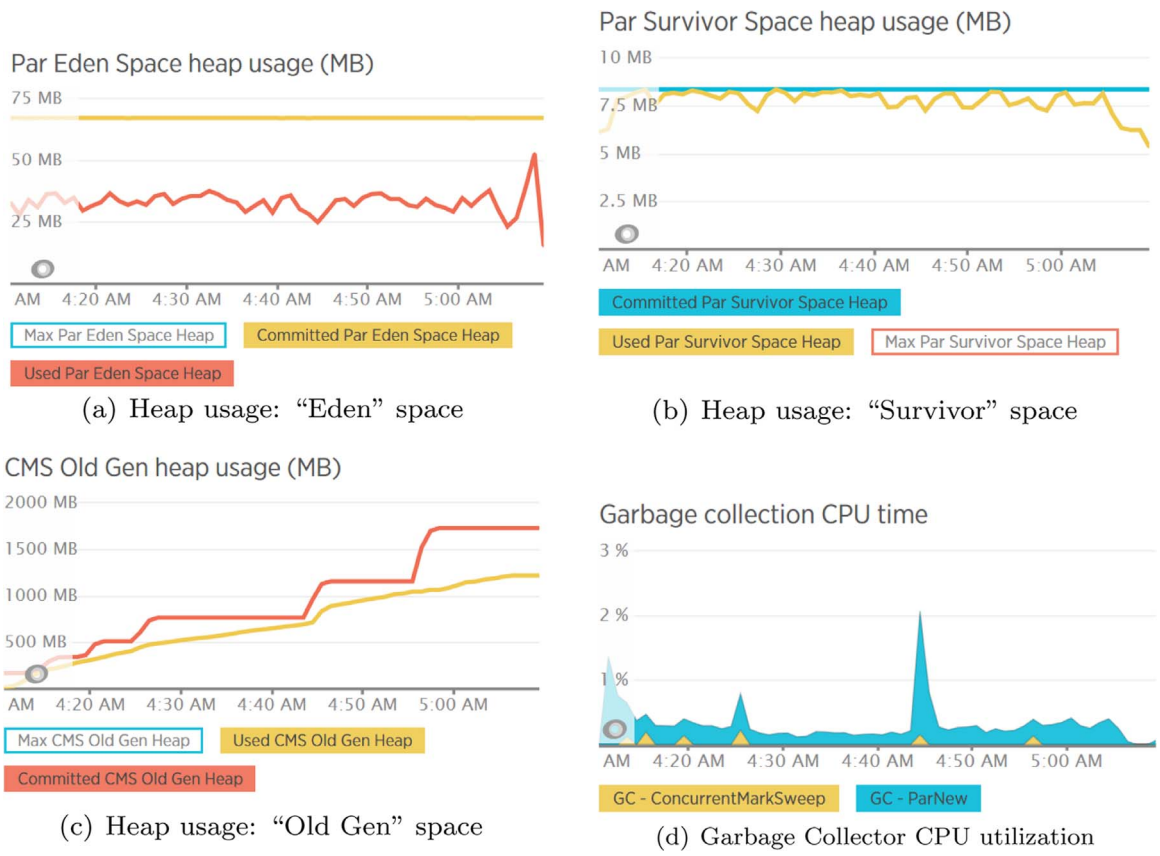


Fig. 17. Heap memory usage: detailed view. (a) Heap usage: "Eden" space, (b) Heap usage: "Survivor" space, (c) Heap usage: "Old Gen" space, (d) Garbage Collector CPU utilization.

Perm (i.e., permanent) Space. The former two categories (New and Old) are managed through the heap, whereas the latter (Perm) is dealt with in the stack. It holds all the reflective data of the virtual machine itself and stores class level details by loading and unloading classes, methods, String pools, etc.. Coming back to the heap, the JVM allocates every generation its own memory pool. It has at least one memory pool and may create or remove memory pools during execution. More precisely, we have the following sub-categories:

1. *Eden Space*, associated with heap new (young) generation: the pool from which memory is 'initially' allocated for most objects;
2. *Survivor Space*, also associated with heap young generation: the pool containing objects that have survived Garbage Collection of the eden space;
3. *Tenured Generation*, associated with heap old generation: the pool containing objects that have existed for some time in the survivor space.

The VM initially assigns all objects to the eden space, and most objects die there. When it performs a minor (or partial) GC, the VM moves any remaining objects from the eden space to one of the survivor spaces. The VM moves objects that live long enough in the survivor spaces to the "tenured" space in the old generation. When the tenured generation fills up, there is a full Garbage Collection that is often much slower because it involves all 'live' objects. From the analysis of the graphs in Fig. 17 we can clearly identify the evolution of memory allocation during the trial in question. Basically, due to the high number of active objects associated with the on-line tracking of monitored calls within Kerberos, most of the dynamically allocated objects keep on being active for a long time, hence causing a double transition form the eden space (Fig. 17(a)) to the survivor space (Fig. 17(b)) and then to the tenured, or Old Gen, space (Fig. 17(c)) which keeps on growing at a fairly sustained rate. This causes the garbage collector to perform, on a regular basis, i.e., every time the previously committed Olg Gen heap allocation becomes insufficient (Fig. 17(d)), a full garbage collection.

7. Conclusions and future work

In this paper we presented Kerberos, a system for real-time detection of frauds in next-generation VoIP networks. Kerberos has been designed and implemented with a well defined set of requirements in mind, ranging from scalability to adaptability to dynamic operational scenarios. The system leverages complex event processing techniques, combined with components separation and asynchronous communication, in order to achieve good performance in the presence of a significant stream of call-related events. The system is currently operating "on stage" inside Tiscali's infrastructure and is almost ready to go into production.

Many are the possible improvements to the system, which all represent the subject of our future work. We are indeed improving both the detection and the remediation parts of the system. With respect to the former, and based on the experiences some of us have made in the past in the field of anomaly-based intrusion detection, we are focusing on improving the capability of the system to self-configure by somehow 'learning' from the traffic patterns observed during its past operation. Coming to remediation, we have recently started to devise solutions aimed at effectively dealing with an ongoing fraud while at the same time retrieving as much information as possible from it. In particular, apart from somehow standard black-listing strategies, we are investigating the adoption of call diversion solutions whereby fraudsters' calls are sent to an advanced honeypot component which tries to gather as many details as possible from them. Information collected at the honeypot side is then processed and exploited in order to further improve the detection part of the system.

References

Abdallah, A., Maarof, M.A., Zainal, A., 2016. Fraud detection system: a survey. J. Netw. Comput. Appl. 68, 90–113. http://dx.doi.org/10.1016/j.jnca.2016.04.007.

Aziz, A., Hoffstadt, D., Rathgeb, E., Dreibholz, T., 2014. A distributed infrastructure to analyse SIP attacks in the Internet. In: Networking Conference, 2014 IFIP, pp. 1–9. http://dx.doi.org/10.1109/IFIPNetworking.2014.6857088.

Chiappetta, S., Mazzariello, C., Presta, R., Romano, S.P., 2013. An anomaly-based approach to the analysis of the social behavior of VoIP Users. Comput. Netw. 57 (6), 1545–1559. http://dx.doi.org/10.1016/j.comnet.2013.02.009, (URL ⟨http://dx.doi.org/10.1016/j.comnet.2013.02.009⟩).

Crane, D., McCarthy, P., 2009. What Are Comet and Reverse Ajax?. In: Comet and Reverse Ajax, Apress, pp. 1–9. URL ⟨http://dx.doi.org/10.1007/978-1-4302-0864-81⟩

De Lutiis, P., Lombardo, D., 2009. An innovative way to analyze large ISP data for IMS security and monitoring. In: . In: Proceedings of the 13th International Conference on Intelligence in Next Generation Networks ICIN, pp. 1–6. doi:10.1109/ICIN.2009.5357065.

Gruber, M., Schanes, C., Fankhauser, F., Grechenig, T., 2013. Voice calls for free: How the black market establishes free phone calls – Trapped and uncovered by a VoIP honeynet. In: Privacy, Security and Trust (PST), 2013 Proceedings of the Eleventh Annual International Conference on, pp. 205–212. http://dx.doi.org/10.1109/PST.2013.6596055.

Hoffstadt, D., Rathgeb, E., Liebig, M., Meister, R., Rebahi, Y., Thanh, T., 2014. A comprehensive framework for detecting and preventing VoIP fraud and misuse. In: Proceedings of the International Conference on Computing, Networking and Communications (ICNC), pp. 807–813. http://dx.doi.org/10.1109/ICCNC.2014.6785441.

Kapourniotis, T., Dagiuklas, T., Polyzos, G., Alefragkis, P., 2011. Scam and fraud detection in VoIP Networks: Analysis and countermeasures using user profiling. In: 50th FITCE Congress (FITCE), pp. 1–5. http://dx.doi.org/10.1109/FITCE.2011.6133427.

Luckham, D., 2008. The power of events: An introduction to complex event processing in distributed enterprise systems. In: N. Bassiliades, G. Governatori, A. Paschke (Eds.), Rule Representation, Interchange and Reasoning on the Web, Vol. 5321 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 3–3. http://dx.doi.org/10.1007/978-3-540-88808-6_2. URL ⟨http://dx.doi.org/10.1007/978-3-540-88808-6_2⟩

McGovern, J., Sims, O., Jain, A., Little, M., 2006. Event-driven architecture. In: Enterprise Service Oriented Architectures, Springer Netherlands, pp. 317–355. URL ⟨http://dx.doi.org/10.1007/1-4020-3705-8 8⟩

Rozsnyai, S., Schiefer, J., Schatten, A., 2007. Solution architecture for detecting and preventing fraud in real time. In: Digital Information Management, 2007. ICDIM '07. In: Proceedings of the 2nd International Conference on, vol. 1, pp. 152–158. http://dx.doi.org/10.1109/ICDIM.2007.4444216.

Uberti, J., Jennings, C., 1998. Usage of cause and location in the digital subscriber signalling system no. 1 and the signalling system No. 7 ISDN user part, ITU-T recommendation Q.850 Q.850. Int. Telecommun. Union, URL ⟨https://www.itu.int/rec/T-REC-Q.850-199805-I/en⟩.

Luca Manunza has a long-track record of experiences in the field of research and development, both in the industry and within the context of research-oriented institutions. Since 2009 he works for Tiscali, where he currently leads the Services and Application Development business unit. He is also the CTO of Tiscali's Streamago project.

Stefano Marseglia received an advanced degree in Computer Engineering from the University of Napoli in 2015. He discussed a thesis on real-time event processing frameworks.

He currently works for a research-oriented startup with a focus on cloud infrastructures and micro services based systems.

Simon Pietro Romano is an Associate Professor in the Department of Electrical Engineering and Information Technology (DIETI) at the University of Napoli. He teaches Computer Networks, Network Security and Telematics Applications. He actively participates in IETF standardization activities, mainly in the Applications and Real-Time (ART) Area.