GenLM: License Management for Grid and Cloud Computing Environments

Mathias Dalheimer and Franz-Josef Pfreundt
Fraunhofer Institut für Techno- und Wirtschaftsmathematik, Kaiserslautern, Germany
{dalheimer|pfreundt}@itwm.fhg.de

Abstract—Software license management allows independent software vendors (ISVs) to control the access of their products. It is a fundamental part of the ISVs’ business strategy. A wide range of products has been developed in order to address license management. There are, however, only few ongoing works with regard to license management in grid and cloud computing environments.

This paper presents our work on GenLM, a license management solution suitable for these environments. It has been built in order to provide a secure and robust solution for ISVs that want to extend their software usage to these systems. We provide ISVs a toolchain to implement arbitrary software licensing models. At the same time we ensure that licenses are mobile, i.e. they can be used on any resource the user has access to.

I. INTRODUCTION

In the last years, grid computing has become an established tool for scientific computing. Scientists from various disciplines use the grid in order to solve large-scale problems or increase the processing speed. Similarly, cloud computing enables everyone to rent an arbitrary number of computers in order to solve problems. The main idea of cloud computing is that resources are only accessed when they are really needed. The improved cost-effectiveness makes cloud computing especially interesting for small- to medium sized enterprises (SMEs) which cannot afford a dedicated infrastructure of this size.

While various grid middlewares are available, the management of software licenses is not addressed in current infrastructures. The 451 group identified licensing problems to be the biggest issue that hinders grid adoption [1]. A recent report of Armbrust et al. argues equally: license management is one of the ten obstacles for cloud computing to be successful [2]. Software licensing models have been designed without considering a geographically distributed usage. This does not only concern the technical implementation of the license management, but also economic and legal constructs. As we have discovered during our work, the legal constructs are the biggest problems when changing licensing models. Both economic and legal innovations in licensing can only be driven by the independent software vendors (ISVs). The technical rendering of a license management must adapt to these requirements.

On the other hand, the usage of commercial simulation software in cloud computing environments is attractive to SMEs. Without any upfront cost, a user can choose to use an infrastructure to solve a specific problem. But this is currently only true for the compute resources – commercial software licenses can typically not be purchased on demand. Nevertheless, we expect on-demand licensing to be more attractive for both ISVs and users in the future. For example, additional licenses could be requested for a short period of overload, while permanent licenses satisfy the average license usage.

In this work we present GenLM, a license management framework that allows ISVs to manage their license usage in a distributed world. The main idea is to attach the license not to a node or a person but to issue licenses for the input datasets. This way, a user can buy a per-job license and run the job on any suitable resource. We use cryptographic primitives in order to secure the licensing framework and to prevent tampering.

In the next section we derive the requirements of a grid licensing mechanism. Then, we introduce the GenLM approach and describe the overall design and implementation of GenLM. Finally, we discuss related work and conclude with a discussion of our work.

II. REQUIREMENTS OF GRID LICENSING

We must consider three stakeholders when discussing licensing techniques: users, resource providers and ISVs. Users expect the licensing mechanism to be transparent. Problems with the licensing mechanism will yield frustration on the user’s side and must be avoided. Providers, on the other hand, would like to be able to offer an infrastructure where users can select from a wide range of applications. However, since the applications need to be licensed, providers currently offer only a limited selection of software packages.

As outlined above it is critical to meet the requirements of the ISVs. Especially in the area of simulation software the products reflect years of technical experience and significant implementation efforts. Often, a commercial software package contains all the knowhow and the experience of an ISV. Therefore, it is central to the business model of ISVs to protect their software from unauthorized usage. License management issues are central business decisions.

Currently, three software licensing techniques are used:

1) The dongle solution requires a hardware token to be present during program execution. The software checks which features of the software are licensed and grants access based on the communication with a dedicated hardware token.
2) The system fingerprinting solution requires a hardware fingerprint to be sent to the ISV. The fingerprint typically consists of CPU identification numbers and other hardware identifiers. The ISV then creates a license key that unlocks the software for the requested hardware.

3) An alternative to these approaches are networked license servers which are currently used in cluster installations. A license server issues licenses to clients on demand. Arbitrary policies can be implemented within the server: for example, the total number of available licenses must not be exceeded at all times. Often, a license server itself is protected by a hardware dongle – that is, the license server maintains the license features on a dedicated hardware token.

Apart from the solutions listed above, other licensing mechanisms are used as well. These are typically inhouse solutions that are not available publicly.

A. Shortcomings of Available Technologies

The outlined licensing techniques were designed for use within an organization. This reflects the contracts made between the ISV and its customers: a customer buys the software and installs it on local resources in order to use it. In grid and cloud computing environments, this assumption is no longer valid. Customers want to be able to use resources that are not located within their institution.

One approach to deal with this is to use software licenses that are available at remote sites. This poses both legal and economic questions: although it is technically possible to use a remote site’s license, the contract between ISVs and their customers limit usage to local site users.

Another problem arises when SMEs want to use licenses of universities and research centers. These license are typically so-called “research and education”-licenses which must not be used for commercial applications. In current grid installations most resources are operated by universities. If a commercial user submits a job, a scheduler might forward the job to a research site where only research licenses are available. This scenario is clearly not desirable for an ISV. All legal contracts are two-party contracts involving the customer and the ISV - no service provider is known in the legal sphere. Some ISVs also tend to negotiate with every customer separately, yielding different conditions for all customers.

In order to preserve this business practice, we would like to find a technical solution that allows licenses to be “mobile”. A mobile license can be used independently of the physical location. At the same time, it ensures that all legal restrictions are fulfilled.

Technically, the licensing techniques listed above are not suitable for grid and cloud computing environments. This is obvious for the dongle solution since a dongle cannot be transferred to a remote site easily. The hardware fingerprinting solution is also not applicable because the user doesn’t know the characteristics of the hardware running the job.

The licensing server approach is more suitable for distributed environments. Two scenarios are feasible, see figure 1:

The license server can reside on the user’s site (license server A) and on the provider’s site (license server B). The latter case is not different from the typical usage scenario outlined above.

In the other scenario the license server resides at the user’s site. A user job attempts to use it during startup to acquire a license over the internet. However, there are several shortcomings:

1) The license server needs to be accessible over the internet. There is currently no access control built into licensing servers – any user can request licenses from the server. The BEinGRID project developed a solution addressing this [3].

2) License servers are not aware of reservations and queues on remote sites. It is possible for a job to fail because there is no license available during job start. A reservation of licenses for a specific job is currently not possible.

3) A third complication arises because both license server and compute node are typically protected by firewalls and NAT devices translate internal and external IP addresses. A reliable service that works for all compute sites would be difficult to implement.

Despite these technical problems legal reasons often forbid this scenario. In the next section we will gather the requirements for license management techniques in grid environments.

As a third scenario the license server in position A might be used by the resource provider in order to issue unlimited pay-per-use licenses. This is technically feasible, however the trust of the ISV in the provider must be high. The BEinGRID project investigates this possibility [3].

B. Requirements for GenLM

In the German grid project “PartnerGrid” we collaborate with different ISVs in order to find a technical solution for licensing in grid environments. As outlined above, a technical solution must obey the legal and business restrictions that are currently in place. In our work with different ISVs we experienced a wide range of license terms and practices. Our proposed framework must be flexible in order to support the business models of many ISVs. The requirements can be summarized as follows:

1) Support for existing license agreements: in practice it is almost impossible to change legal contracts during their
lifetime. A licensing framework must support existing contracts.

2) On-Demand licenses: in addition to the classical “CPU-per-Year” licenses, it should also be possible for an ISV to offer on-demand licenses when desired. In combination with the first requirement this would allow complex licensing terms: for example, an user must purchase a base license which is valid for a year. Additional software modules could then be licensed on-demand on top of the existing base license.

3) Mobility of licenses: as outlined above, grid providers and cloud computing datacenters are usually not involved in the license agreement. Technically, the license should be valid regardless of the execution location of the computations.

Beside these requirements we also need to fulfill certain non-functional requirements:

1) The framework must be easy to integrate with existing software packages. Ideally, the existing license management routines will be replaced or enhanced by the new framework. The framework must also be portable across different operating systems.

2) Support for both grid and cloud computing environments must be built in. This requires special care regarding network friendliness and security.

3) The system must be highly available. If users can’t request licenses when they demand them, this might have serious implications on the ISV.

4) The system must be secure. Licenses must not be granted without a license contract. Additionally, it must be difficult for malicious crackers to break the license checks in the software itself.

In the next section we introduce our approach to license management.

III. THE GENLM APPROACH

Users, resource providers and ISVs form the three stakeholders of grid computing, as outlined above. We have developed one component for each of the stakeholders, see figure 2. The GenLM client is used by the user or by a preprocessing software to acquire a license token. The GenLM server is responsible for issuing licenses on demand. The GenLM license verifier is included in the ISVs compute software and checks whether a license is valid for the pending job.

The central idea of GenLM is to attach the software license to the input data of the batch job. We create a license token for a given set of input data. The license token is a file that can be transferred together with the input data to the compute site. It contains all information the license verifier needs in order to check the validity of the software license.

A. The License Token Lifecycle

In order to clarify the concept of our license token we will shortly outline the lifecycle of the token. A token consists of a set of hashes of the input files and a license terms specification which is software-specific. The token is generated by the GenLM client and signed by the GenLM server. During job startup the token is evaluated again, see figure 2.

When a user wants to submit a job she needs to acquire a valid license token for her input data. The GenLM client starts by computing a cryptographic hash for all input files. These hashes are stored in the token. By the construction of the hash they uniquely identify the input files. In addition, the license terms requested by the user are stored in the token. These license terms are ISV-specific and would typically contain information such as the number of requested cores, the software modules to be used for the job etc. GenLM doesn’t evaluate these terms – typically, this information is used by the ISV to decide which license to issue for this specific job.

The request token is then signed with the user’s X.509 certificate and sent to the GenLM server. The server extracts the license terms and the user’s identity from the request token. This information is forwarded to a policy plugin which can be implemented by the ISV, see section IV-C. The purpose of the plugin is to enforce the ISVs business model. For example, the user’s identity can be matched against a customer database. Depending on the contract of the user’s organization the request will be billed separately or it is covered by a flat-fee license agreement. All necessary steps in order to be able to bill the customer will be made in the policy plugin. This might involve putting a billing record in a database.

Assuming the license request is granted the server uses its own certificate to sign the request token. It is then sent back to the user. We call the signed request token a license token since it contains a valid license.

Together with the input data the license token is transferred to the compute site. A job is enqueued at the site which will finally compute the results. On job startup the license token is inspected: First, the signature of the token is inspected. The public key of the license server is used to verify the signature of the license token. If the signature is correct, the application computes the input file hashes based on the local files and compares them with the hashes stored in the token. Given that the locally computed hashes are identical to the hashes stored in the license token, we know that the license server granted this job. The computation can start.

B. Cryptographic Algorithms

As outlined above we rely on hashing and signature algorithms. In this section we describe the cryptographic primitives in more detail.

Let $I$ be the set of input files. The request token is a tuple $RT = (LT, H_I)$ with license terms $LT$ and a set of hashes of the input files $H_I$. The hashes are generated with a collision-resistant one-way hash function $h^F$. For all input files $i \in I$ we compute $H_i = h^F(i)$. By construction of the hash function we get a practically unique identification of the contents of file $i$.

When the request token is signed by the GenLM server we rely on asymmetric encryption. The server has a key-pair $(p, s)$ where $p$ is the public key and $s$ is the private key. The key-pair $(p, s)$ must satisfy the requirements of the encryption.
function $e$. In order to sign the request token the server uses its secret key to sign the hash of $RT$:

$$\text{sig}(RT) = e_s(h^S(RT))$$

(1)

This signature is then attached to the original token, giving the license token the form $T = (RT, \text{sig}(RT))$. When the license verifier evaluates a license token it uses the public key $p$ of the server to compare the signature with the locally computed signature hash:

$$h^S(RT)^e = e_p(\text{sig}(RT))$$

(2)

If this equation holds we can guarantee that the token was signed with the server’s private key – otherwise, we reject the job.

Please note that we didn’t check a revocation list for the server’s certificate. This is a compromise we made in order to simplify the deployment of the license verifier. It is often not guaranteed for compute resources to have full access to the internet in order to check a revocation list during the license verification process. Since the license server’s key will not be distributed we assume that the risk of compromised keys is relatively low. It is also possible to revoke keys during updates of the ISVs software packages (e.g. maintenance releases). It is, however, easy to integrate a revocation list check if the need arises.

We decided to keep the protocol independent of the concrete algorithm – this allows us to change the algorithm if weaknesses are discovered. In addition, we only chose algorithms that are recommended by the German “Bundesamt für Sicherheit in der Informationstechnik”. All algorithms are approved for safety-critical use until 2014 [4]. At the moment we are using SHA-256 [5] as the hash-function $h$. The signatures are generated using X.509 certificates with the RSA algorithm [6] in the Digital Signature Standard (DSS) [7] form. We rely on the OpenSSL implementation of these algorithms [8].

Due to the modularity of the server it is also possible to use dedicated hardware encryption devices for generating the signature. Since the server will reside at the ISV’s site we didn’t consider such devices yet.

IV. IMPLEMENTATION

The GenLM software has been implemented using C++. Right now the software targets the Linux environment but it will be ported to Windows as well. An ISV will obtain our SDK which allows the integration of our components in the ISV software. In this section we will outline the implementation choices we made.

A. Protocol Design

When a GenLM client requests a license, the GenLM server performs a complex operation: for example, if an on-demand license is requested, the user will be billed for it. Since we communicate over the internet we cannot assume reliable message transports. Nevertheless we would like to have transaction-like semantics for the licensing mechanism. During the development of the protocol we used model checking methods. We have proven that the protocol is robust with regard to message losses.

In general, one can distinguish simulation and testing, deductive verification and model checking methods for the validation of complex applications [9]. Both simulation and testing observe the output of a system under a given input. In general, it is not possible to test or simulate all possible inputs, thus no complete verification is possible. In contrast, deductive and model verification allow a complete verification of a given model. Deductive verification is mostly a manual process and therefore time-consuming.

For our work, we choose model verification as verification technique. Model checking takes a model and a specification as input and checks whether the model fulfills the specification at all times [10]. If not, a trace is given which allows the user to reconstruct the behaviour of the system leading to the specification violation.

We use SPIN to verify our model [11]. The models are specified in a C-like language called PROMELA. A PROMELA model consists of the definition of processes and the messages which are exchanged through FIFO pipelines. This makes SPIN a good choice for the verification of distributed systems - SMV [12] for example employs data transfer via shared variables, which is not a good match for communication protocols. SPIN translates the model in finite state automata and enumerates all possible states of the whole model. It is possible to check for liveliness and deadlocks. In addition, one can check specifications written as linear temporal logic formulae.

Our model consists of three processes: The client part, the server part and a so-called “lossy daemon”. In order to simulate the unreliable nature of the internet the lossy daemon consumes messages randomly from the client-server communi-
cation. The client and server parts fulfill the objectives outlined above.

The model has been verified completely without any restriction of the state-space. Our initial design followed the simple request-reply schema: the client requests a license and the server replies with the signed token. But during model checking it turned out that the signed token can get lost during transfer – this potentially bills a customer but leaves him without a valid license. Even worse, a customer can state that no license was delivered although the license was correctly transmitted. The customer could use the license without paying for it. This can happen if the network looses certain messages, see figure 3.

Instead we introduce another request-reply message exchange into the protocol, see figure 4. After requesting a signed token the client receives an encrypted version of the signed token. The server passes a key identifier along with this message. The client then requests the decryption key from the server using its key identifier. When the server receives the valid key identifier it knows that the token has been received correctly. The billing process can start now. This procedure follows the principles of two-phase commit protocols [14].

The protocol does not contain any dead- or livelocks. We also modeled certain properties of the protocol explicitly in form of linear temporal logic (LTL) formulas. In order to simplify the verification, we introduced predicates for both the client and the server, see table I. We use the symbol \( C \) for the client and \( S \) for the server. Each process updates the value of its predicate variables according to the messages it receives.

It is now easy to verify properties based on the values of these variables. Amongst others we have proven the following properties\(^1\):

1) If the client has received the decryption key it was also billed by the server:

\[
\square (C_k \rightarrow (\diamond S_b))
\]

2) If the client could use the license, the server must bill the client eventually.

\[
\square (C_k \rightarrow (\diamond S_b))
\]

3) If the server does not bill, the client never receives the decryption key.

\[
\square (S_b \rightarrow (\diamond C_k))
\]

4) If the server sends the decryption key, the client will be billed eventually.

\[
\square (S_k \rightarrow (\diamond S_b))
\]

\(^1\)We use the “standard” symbols for temporal logic: \( \square p \) means “always \( p \)” - the term \( p \) is an invariant. \( \diamond p \) means “eventually \( p \)” and describes the guarantee that \( p \) will hold in the end.

---

Fig. 3: When the network consumes messages, undesired behavior might arise. In this example the license token is consumed by the lossy daemon – the customer doesn’t receive his license but is billed for it.

Fig. 4: A successful message exchange. Please note that although no message losses occur the protocol allows all messages to get lost.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_L )</td>
<td>The client has received a full license.</td>
</tr>
<tr>
<td>( C_K )</td>
<td>The client obtained the decryption key.</td>
</tr>
<tr>
<td>( S_K )</td>
<td>The server has sent the decryption key.</td>
</tr>
<tr>
<td>( S_B )</td>
<td>The server billed the client.</td>
</tr>
</tbody>
</table>

TABLE I: The predicate definitions used during the verification of the protocol.
B. Architecture of the Implementation

The cryptographic algorithms are rather time-consuming. At the beginning of the system design we decided to use an architectural approach that would let us use multiple CPU cores in order to maximize the parallelism of the operations. We decided to structure the system using the Staged Event-Driven Architecture (SEDA) as described by Welsh [15]. A SEDA program is composed of stages that act independently. A stage consists of an input queue and a thread pool. The threads dequeue a work item from the input queue and apply the stage strategy. The strategy defines the algorithms to be applied to the work items.

In our case the system architecture looks as in figure 5. We connect to a message queue in order to receive and send messages. A channel-adapter stage receives the messages and forwards them to a decoding stage. This stage builds an internal representation of the external messages from the wire format. The internal message representations are forwarded to the application logic stage. This is the most complex stage: It contains the state machine for all sessions and some bookkeeping facilities. It can also use a separate cryptographic stage to sign tokens. The application logic queries a policy plugin in order to decide whether to grant a license, see section IV-C. When a message needs to be sent to the clients it will be forwarded to the encoding stage which produces the wire format. The channel-adapter stage submits the message to the destination peer.

Another advantage of using the SEDA approach is simplified testing. Since the stages are only loosely coupled we can test them independently. This allows us to maintain a testsuite based on unit tests. In combination with the protocol verification we can construct tests of all aspects of the system. We can also simulate message sequences that would be very hard to produce using a tightly coupled system.

We communicate using a messaging queue, in our case Apache ActiveMQ [16]. ActiveMQ supports a wide range of language clients and protocols. It allows for high availability and load balancing deployments. We can built highly reliable license servers on top of it. Please note that there is no dependency on any grid or cloud middleware.

C. Policy Plugin

The policy plugin allows ISVs to implement their licensing policies. The plugin will be queried whenever a license request needs to be processed. We provide an interface that can be used in several ways: The ISV’s policy can be compiled into the license server, but it is also possible to use a Ruby module which is interpreted on the fly.

A policy plugin consists of a set of routines that will be called by the application logic stage at different states of the protocol. One can put billing information in a database whenever a license is issued successfully. We also rely on a database in order to store the decryption keys.

As outlined above it is not feasible to identify ISV licensing schemes in advance. We offer consulting services in order to adjust the policy plugins such that they fulfill the ISVs needs. A cost-based accounting can be built on top of our system.

D. Performance Considerations

The whole GenLM toolsuite is built around strong cryptographic primitives. It is also a network server infrastructure – the typical performance characteristics of a network server apply. In order to evaluate the scalability of the toolsuite we need to examine networking and cryptographic aspects.

Network servers are typically IO-bound. When the load increases one expects the response time to increase. The service should gracefully level off. The SEDA architectural approach has been designed with these aspects in mind. The architecture is known to behave well-conditioned under high loads [17]. Additionally we can utilize modern multi-core machines without further synchronization penalty.

In order to scale the service to more than one server we use functions of the underlying message queue. Since the client requests are independent of each other we can simply add more servers to a pool in order to scale the service horizontally. The message queue allows a natural load balancing: servers do only poll for new messages if they have the capacity to handle the load. When several servers work together, they distribute the load evenly according to their capacity. Please note that we use the message queue also to map one client’s requests to the same server.

When a user wants to acquire a license the GenLM client needs to compute a hash function for the input data set. We conducted a series of experiments in order to evaluate the performance of our request token generator. We generated a set of random input files with the sizes 100MB, 1GB and 10GB, respectively. These approximate the input dataset sizes we have observed for different real applications. All benchmarks were executed on our institutes “Hercules” cluster system. A compute node consists of a dual Intel Xeon 5148LV ("Woodcrest") with 8GB RAM and a local harddisk of 80GB. We used two different filesystems, namely a GPFS filesystem with 450 MB/s throughput and the local harddisk with 55 MB/s throughput.

Generally, SHA256 is the slowest hashing function we tested. Although MD5 and SHA1 are significantly faster they should not be used in real deployments for security reasons. Especially the MD5 hash can not be regarded collision-free any more [18] [19].

For the local disk the 100MB and 1GB measurements can be served from the filesystem cache of the cluster node, see figure 6a. For 10GB, the dataset size exceeds the 8GB RAM of the node, therefore flushing the filesystem caches. The local harddisk limits the throughput to approx. 55 MB/s.

The whole GenLM toolsuite is built around strong cryptographic primitives. It is also a network server infrastructure – the typical performance characteristics of a network server apply. In order to evaluate the scalability of the toolsuite we need to examine networking and cryptographic aspects.

Network servers are typically IO-bound. When the load increases one expects the response time to increase. The service should gracefully level off. The SEDA architectural approach has been designed with these aspects in mind. The architecture is known to behave well-conditioned under high loads [17]. Additionally we can utilize modern multi-core machines without further synchronization penalty.

In order to scale the service to more than one server we use functions of the underlying message queue. Since the client requests are independent of each other we can simply add more servers to a pool in order to scale the service horizontally. The message queue allows a natural load balancing: servers do only poll for new messages if they have the capacity to handle the load. When several servers work together, they distribute the load evenly according to their capacity. Please note that we use the message queue also to map one client’s requests to the same server.

When a user wants to acquire a license the GenLM client needs to compute a hash function for the input data set. We conducted a series of experiments in order to evaluate the performance of our request token generator. We generated a set of random input files with the sizes 100MB, 1GB and 10GB, respectively. These approximate the input dataset sizes we have observed for different real applications. All benchmarks were executed on our institutes “Hercules” cluster system. A compute node consists of a dual Intel Xeon 5148LV ("Woodcrest") with 8GB RAM and a local harddisk of 80GB. We used two different filesystems, namely a GPFS filesystem with 450 MB/s throughput and the local harddisk with 55 MB/s throughput.

Generally, SHA256 is the slowest hashing function we tested. Although MD5 and SHA1 are significantly faster they should not be used in real deployments for security reasons. Especially the MD5 hash can not be regarded collision-free any more [18] [19].

For the local disk the 100MB and 1GB measurements can be served from the filesystem cache of the cluster node, see figure 6a. For 10GB, the dataset size exceeds the 8GB RAM of the node, therefore flushing the filesystem caches. The local harddisk limits the throughput to approx. 55 MB/s.

The whole GenLM toolsuite is built around strong cryptographic primitives. It is also a network server infrastructure – the typical performance characteristics of a network server apply. In order to evaluate the scalability of the toolsuite we need to examine networking and cryptographic aspects.

Network servers are typically IO-bound. When the load increases one expects the response time to increase. The service should gracefully level off. The SEDA architectural approach has been designed with these aspects in mind. The architecture is known to behave well-conditioned under high loads [17]. Additionally we can utilize modern multi-core machines without further synchronization penalty.

In order to scale the service to more than one server we use functions of the underlying message queue. Since the client requests are independent of each other we can simply add more servers to a pool in order to scale the service horizontally. The message queue allows a natural load balancing: servers do only poll for new messages if they have the capacity to handle the load. When several servers work together, they distribute the load evenly according to their capacity. Please note that we use the message queue also to map one client’s requests to the same server.

When a user wants to acquire a license the GenLM client needs to compute a hash function for the input data set. We conducted a series of experiments in order to evaluate the performance of our request token generator. We generated a set of random input files with the sizes 100MB, 1GB and 10GB, respectively. These approximate the input dataset sizes we have observed for different real applications. All benchmarks were executed on our institutes “Hercules” cluster system. A compute node consists of a dual Intel Xeon 5148LV ("Woodcrest") with 8GB RAM and a local harddisk of 80GB. We used two different filesystems, namely a GPFS filesystem with 450 MB/s throughput and the local harddisk with 55 MB/s throughput.

Generally, SHA256 is the slowest hashing function we tested. Although MD5 and SHA1 are significantly faster they should not be used in real deployments for security reasons. Especially the MD5 hash can not be regarded collision-free any more [18] [19].

For the local disk the 100MB and 1GB measurements can be served from the filesystem cache of the cluster node, see figure 6a. For 10GB, the dataset size exceeds the 8GB RAM of the node, therefore flushing the filesystem caches. The local harddisk limits the throughput to approx. 55 MB/s.

The whole GenLM toolsuite is built around strong cryptographic primitives. It is also a network server infrastructure – the typical performance characteristics of a network server apply. In order to evaluate the scalability of the toolsuite we need to examine networking and cryptographic aspects.

Network servers are typically IO-bound. When the load increases one expects the response time to increase. The service should gracefully level off. The SEDA architectural approach has been designed with these aspects in mind. The architecture is known to behave well-conditioned under high loads [17]. Additionally we can utilize modern multi-core machines without further synchronization penalty.

In order to scale the service to more than one server we use functions of the underlying message queue. Since the client requests are independent of each other we can simply add more servers to a pool in order to scale the service horizontally. The message queue allows a natural load balancing: servers do only poll for new messages if they have the capacity to handle the load. When several servers work together, they distribute the load evenly according to their capacity. Please note that we use the message queue also to map one client’s requests to the same server.

When a user wants to acquire a license the GenLM client needs to compute a hash function for the input data set. We conducted a series of experiments in order to evaluate the performance of our request token generator. We generated a set of random input files with the sizes 100MB, 1GB and 10GB, respectively. These approximate the input dataset sizes we have observed for different real applications. All benchmarks were executed on our institutes “Hercules” cluster system. A compute node consists of a dual Intel Xeon 5148LV ("Woodcrest") with 8GB RAM and a local harddisk of 80GB. We used two different filesystems, namely a GPFS filesystem with 450 MB/s throughput and the local harddisk with 55 MB/s throughput.

Generally, SHA256 is the slowest hashing function we tested. Although MD5 and SHA1 are significantly faster they should not be used in real deployments for security reasons. Especially the MD5 hash can not be regarded collision-free any more [18] [19].

For the local disk the 100MB and 1GB measurements can be served from the filesystem cache of the cluster node, see figure 6a. For 10GB, the dataset size exceeds the 8GB RAM of the node, therefore flushing the filesystem caches. The local harddisk limits the throughput to approx. 55 MB/s.

The whole GenLM toolsuite is built around strong cryptographic primitives. It is also a network server infrastructure – the typical performance characteristics of a network server apply. In order to evaluate the scalability of the toolsuite we need to examine networking and cryptographic aspects.

Network servers are typically IO-bound. When the load increases one expects the response time to increase. The service should gracefully level off. The SEDA architectural approach has been designed with these aspects in mind. The architecture is known to behave well-conditioned under high loads [17]. Additionally we can utilize modern multi-core machines without further synchronization penalty.
Fig. 5: Overview of the SEDA stages in the server. The ChannelAdapter connects the server to the message queue. Encoding and decoding of messages are handled in dedicated stages, as well as the cryptographic functions. The application logic communicates with the policy plugin and implements the protocol.

Fig. 6: Average throughput of different hashes. The local filesystem performance degrades after as soon as the data doesn’t fit into the local machine’s caches. SHA256 shows the lowest throughput in all cases.

We expect that the observed 80MB/s throughput for SHA256 are sufficient for our targeted deployment scenarios - usually, the local disk performance will not allow to increase the throughput. Ferguson et al. submitted the Skein hash function family to the SHA-3 competition [20]. These hash functions show a promising performance, but further evaluation with regard to the security properties are needed before we consider to use them.

E. Security Considerations

A software licensing product must be secure with regard to various aspects. As outlined above the protocol itself must be fault-tolerant. Customers must not be billed for unusable licenses. At the same time, we must prevent customers to acquire licenses without being billed for.

The only entity generating licenses is the GenLM server. It usually resides at the ISVs site with no direct access to the outside. Since the communication is handled by the message queue, only indirect messages from the internet can reach the server. Direct manipulation of the server and the X.509 keypair are not possible.

The GenLM client can also not be manipulated since its only purpose is to request a license. It merely forwards the license token to the resource provider’s site. The GenLM license verifier checks the input data and the token during job startup. Essentially this is an if statement: if the license is valid then proceed, abort otherwise. If an attacker can identify the if statement it would be possible to change the jump addresses. This is basically true for all license management techniques – even a hardware dongle solution relies on such an if statement.

In practice one can use code obfuscation techniques to make it hard for attackers to identify the if-statement in the compiled code. It is also possible to include a code signature to detect modified code. The Microsoft Authenticode technology provides a complete toolchain for this [21].

In addition to the protection mechanisms above we also think that the distribution of the ISV’s software will be limited to service providers. They will be able to obtain a cloud computing license in order to provide the software to their customers without having a license on their own. This limits the exposure of the software to a wider public.

V. RELATED WORK

A variety of commercial license management products is available on the market. The most well known include Acesso’s FLEXnet [22], IBM’s LUM [23], HP’s iFOR/LS
Currently, two European projects develop grid-aware licensing solutions. The BEinGRID project provides a license tunneling mechanism for FlexLM-based products. In addition, they assume that ISVs will need to support pay-per-use licensing models in the future [3]. The project investigates possible business models and legal consequences for ISVs. The SmartLM project [26] builds a software management solution based on WS-Agreement [27]. A license contract can be (re-)negotiated during various stages of the job lifecycle. Li et al. reviewed the requirements for license management in grid and cloud computing environments [28]. They argue that licenses should be managed in terms of a service level agreement and scheduled by a scheduler. We disagree, because it is not in the interest of the ISVs to have their resource usage optimized by a scheduler. There is no incentive for ISVs to follow such an approach.

VI. CONCLUSION

In this paper we presented our work on GenLM, a license management framework for virtualized environments. The key idea is to bind an on-demand license to the input data of a given job. Then, both data and license can be moved to arbitrary execution locations with no dependency on the environment. The implementation is based on strong cryptographic primitives and the client-server protocol has been verified with regard to certain properties. The requirements listed in section II-B have been fulfilled. Obviously, licenses are mobile and can be created on-demand. Existing license agreements can be wrapped in a policy plugin, which decides license requests based on the underlying agreement. The system is secure, cryptographic algorithms can be replaced if they are compromised. Although not presented in this paper, high availability can be achieved by using features of the message queue. It is also easy to integrate GenLM in existing software products.

Please note that while a patent is pending for this solution we have fully disclosed our technology. Our product is currently in a closed beta test and will be publicly available in the first quarter of 2009. We are currently targeting the Linux platform, but we will port the software to Windows as well. We will keep both versions interoperable: It will be possible to request a license on a Windows machine while both the license server and the license verifier run on Linux.

ACKNOWLEDGMENT

The authors would like to thank the Fraunhofer ITWM for the continuous support. This work was supported by the German “Bundesministerium für Bildung und Forschung” under contract no. 01G07009A-D. We also thank Alexander Petry for valuable discussions and feedback.

REFERENCES