Pay as You Go – Associating Costs with Jini Leases

Peer Hasselmeyer Markus Schumacher Marco Voß
Information technology Transfer Office,
Darmstadt University of Technology,
Wilhelminenstr. 7, 64283 Darmstadt, Germany
{peer,schumacher,mavoss}@ito.tu-darmstadt.de

Abstract

Jini is a technology for building reliable, fault-tolerant distributed applications. Besides offering an infrastructure for clients to locate services, it introduces the concept of leasing. Leases model time-constrained access granting and are used for distributed garbage collection. In this position paper we propose an extension to the lease concept that allows associating costs with resource access. To make this extension usable in a commercial environment authentication, encryption, and non-repudiation are required. We analyze our extension in this respect and identify the types of security needed. Based on this analysis we introduce an architecture consisting of a number of Jini-enabled services that support non-repudiation and accounting.

1. Introduction

The Jini connection technology [18, 14] is an innovative and usable technology for building reliable, fault-tolerant distributed applications. It provides an infrastructure that allows clients to find services independent of both parties' location. The dynamic nature of locating and using services is one of Jini's major strengths. It is the base for the creation of plug and play devices and services.

But Jini is not only about finding services. The Jini specifications also define a few other concepts that are of major importance for reliable distributed applications. Besides distributed events, leases [15] are an important paradigm. Although originally introduced for distributed file cache consistency [3], within Jini they are primarily used for distributed garbage collection. Resources are allocated for only a limited amount of time. If no interest in a resource is expressed after that amount of time, the resource can be freed and reused. Resources can include CPU cycles, storage, event notifications, name service entries, etc.

Looking at the scenarios [17] that Jini is aimed at, we found that a number of services to be offered within a Jini federation are time-based and are likely to be offered by a third party, especially when on the move. Third parties supplying services to clients must get revenue in some way. On the world-wide web this is often done by placing advertisements on web pages. In a Jini system this model is not appropriate as commercials are aimed at human consumption. In a Jini federation, though, only software entities talk to each other. We therefore anticipate that the use of Jini services will be charged directly to the service user's bill. As Jini already has the notion of leases, it is an obvious choice to extend the given framework with the ability to associate costs with leases.

Charging a certain amount of money per time unit is one (easy) thing, but to make it usable in a commercial environment a large number of related topics has to be addressed. Contracts between the service provider and his clients have to be made, service usage has to be metered and billed, and there must be proofs for service requests, request acceptance, renewal request, etc. In this paper we focus on an extension of the Jini leasing infrastructure and all problems that are directly related to starting and terminating time-constrained service access. It will be shown how the use of non-repudiation methods can solve problems related to proving service requests and access granting. We do not address other closely related areas, like the negotiation of contracts, or proving service provisioning.

In this paper we will take a look at some problems that are related to the introduction of "billable leases". Problems dealing with accounting and proofs of service usage are identified and solutions are proposed. Section 2 briefly describes the current Jini leasing concept. Some scenarios that we are thinking of are introduced in Section 3. These scenarios are then analyzed and required security measures are identified in Section 4. Section 5 presents required background on non-repudiation. In Section 6 we go on to present an architecture that addresses the identified problems and that is able to support the described scenarios. Our prototypical implementation is described in Section 7. Other relevant work is evaluated in Section 8 and we finally give an outlook in Section 9 on what else has to be done to make billable leases become reality.

2. The Jini leasing concept

A Jini lease is a contract between two parties. One party, the lease grantor, is willing to grant some of its resources to the other party, the lease holder. The contract is valid for a limited amount of time only. Before granting access to the resource, the amount of time is negotiated between the two parties where the grantor has the final word on the lease's length. Upon expiry of a lease, the associated resource can be freed by the lease grantor and might be used otherwise, e.g., assigned to another client. If a lease holder wants to extend the period of time it is allowed to use a resource, it can ask the lease grantor to renew the lease and set a new expiration time. Here, too, the lease grantor has the final word on the expiration time of the lease including the possibility of not extending the lease. If a lease holder is finished using a resource before the associated lease ends, rather than waiting for the lease's expiry it can explicitly cancel the lease.

The expiration of a lease might not only be due to a client not needing a resource anymore, but also because communication between the lease grantor and the lease holder is lost. Reasons include broken network links and crashed clients. In case of a catastrophic event at the client, the resource allocation might be forgotten and is therefore never explicitly released. In a simple-minded distributed application the resource might therefore be reserved forever, or at least until an administrator frees the resource by hand. Besides this being a tedious task, it leads to ineffective resource utilization. In a Jini federation, forgotten resource allocations are eventually removed automatically when the associated lease expires. The duration of a lease is therefore an important trade-off between network traffic (for lease renewal) and unnecessary resource allocation. As stale registrations can be considered garbage, leases are a means to enable distributed garbage collection.

The way how a client acquires a lease (and access to the associated resource) is not defined by the Jini lease specification. A common scenario starts with a client asking a service provider for access to some resource for a desired amount of time. If the resource is available, the service provider grants access and returns—possibly in conjunction with other data—a lease to the client. Further communication about extensions or cancellations of the lease are performed via the lease object.

The Jini specifications include the basic Java interfaces for leasing. Most important, they include the `Lease` interface itself which is shown in Figure 1. Objects implementing this interface are generated by the lease grantor and sent to the client upon granting access to a resource. Besides

```
public interface Lease {
  long getExpiration();
  void cancel() throws UnknownLeaseException,
                       RemoteException;
  void renew(long duration)
         throws LeaseDeniedException,
         UnknownLeaseException, RemoteException;
  [...]
}
```

Figure 1. The `Lease` interface (abbreviated)

other functions not shown and discussed here, the interface contains methods that allow the lease holder to ask for the expiration time, lease renewal, and lease cancellation. All time values are represented by `long` values with a unit of 1 millisecond.

Within the Jini infrastructure, leases are used for distributed garbage collection. Whenever a connection between entities of a Jini federation probably lasts longer than for just a single function call, resource access is leased. Service registrations with a Jini lookup service [16], event registrations, and transactions are leased. As an example, we can take a short look at how leases are used with distributed events. Registrations for event delivery are associated with a lease. Event generators notify interested clients only as long as the clients renew their leases. Event generators therefore have a method to register clients. The object returned by this method is a lease. Event generators do not have a method to unregister clients, though, as this can easily be achieved by canceling the lease.

3. Leasing scenarios

When buying services there are two basic models for usage accounting: unit-based and time-based. As an example, take a client who wants to access a document database. The service provider might charge an amount for each document retrieved. This corresponds to the unit-based model. When using a time-based model, the client might pay a certain amount of money to get access to the database for a limited period of time. In that period of time the client can access as many documents as she wants. Depending on the service considered, one of the models might be more appropriate. Even mixed models are possible, e.g., where a client pays a fixed basic rate plus some additional amount for each service access. As leases always model a time-constrained relation, we do only consider time-based accounting models in this paper.

A good example for a leased service is a storage service. Many applications, e.g., data visualization, generate large amounts of temporary data that is only needed during computation but is discarded when the computation is finished. It is not economical to buy additional disk capacity that is needed for only a short amount of time. Instead, the necessary capacity can be leased from a storage service for the time needed.

Another common leasing scenario is a software component that is only needed infrequently, maybe once a year to prepare the annual accounts. It is not only cheaper to lease the software for the amount of time needed than to buy it, it also saves the time for maintaining the software and keeping it up-to-date.

Before using a service—which is modeled by a lease—usually a contract has to be negotiated between the service provider and the service user. This contract has to include at least data about the price of the service, the kind of service that is provided, and the quality of the provided service. There are many ways in which a contract can be established. This could be done by humans signing a physical contract and just putting a proxy into the system. Also a completely electronic contract is possible, e.g., by using TPAML [12]. It is important to note that a lease has to be linked to a contract in some way. This question is further discussed in Section 6.

An interesting point is that in some cases a contract might not even be needed. Just think of the long distance access code scheme: you can choose whichever telecommunications provider you want without explicitly establishing a contract. Basically, the contract is automatically established when you dial the provider's prefix. This scheme could very well be used when accessing services. Leases that "know" the current cost could prove to be helpful in this scenario.

Another issue is how service usage will be paid for. Several solutions seem to be possible. If a contract is established between the two business partners, it will usually contain terms about the method of payment. In case of service access by a private customer, an easy method of payment would be to charge the customer's credit card. An even better method for paying small sums would be to use some kind of electronic cash.

4. Risc analysis

When we analyze the scenarios that are described in the previous section, we determine several threats that can result in security breaches. Note that it is irrelevant whether such a violation against the security guidelines is done on purpose or by accident. We think that the following threats are the most critical ones in our context:

- Violation of Integrity: modification or destruction of messages/function calls.
- Theft of Information: disclosure of confidential data, e.g. by eavesdropping or spoofing.
- Denial of Service: prevent authorized clients from using a service, e.g. by flooding.
- Repudiation: it cannot be verified that a user has performed certain actions, e.g. that he has sent a message.

As our paper focuses on associating costs with Jini leases, we want to take a closer look at the last item. In a trusted environment a service provider must be sure that a client has really ordered a service. It should be impossible to use a service without payment. On the other hand, the client wants to be sure that the service provider has received the order. Furthermore, it should be impossible that a service provider tries to get money for a service that he has never provided. We do not address the last problem in this paper. Although our architecture allows the generation of evidence for service provisioning, proving that a service was really provided is usually application-dependent and sometimes hard to decide without human intervention, especially if the service has effects on the real, physical world. We therefore only address the problem of generating evidence for requesting and agreeing to starting and terminating service access. How to prove actual service usage is not part of this paper.

Appropriate countermeasures against the threats described above have to be applied in order to limit or prevent potential losses:

- Authentication of clients: service access costs have to be billed to the requesting client. She therefore has to be identified correctly.
- Authentication of servers: in case of a dispute, the service provider must be known to the client. Hence, he has to be identified correctly, too.
- Non-Repudiation: in order to settle disputes, evidence of certain user actions has to be provided.

Anonymity can be an important issue in electronic market places, but contradictory to the countermeasures described above. It could be achieved by additional proxy services which can be used by clients and service providers. Then, only the proxy services know the real identity of their corresponding clients and act on their behalf. Certainly this additional level of indirection adds more complexity to the non-repudiation protocol.

Another solution to keeping the identity of the client unknown is the use of electronic cash. As said before, the client's identity is needed to charge him for service access. If he pays by e-cash upon service access, billing data is not needed and can therefore be kept secret. On the downside, the client does not have any usefull evidence in case of a dispute. Although he has a proof that the service provider accepted his service request, the proof does not include the client's identity. He can therefore not prove that it was him who requested the service.

Integrity and confidentiality can be provided by a secure communication channel. Although not mentioned explicitly until now, access control is another important security feature that has to be provided in a secure Jini environment.

5. Concepts for non-repudiation

The idea behind non-repudiation is that the parties of a contract want to have some kind of guarantee that each fulfills its part of the contract. In case of a dispute they can give the contract to an arbitrator that decides who is wrong and who is right. To clarify this, we make use of some real world examples:

1. A buys a DVD player from a dealer B and pays on cash terms. A receives a signed bill for cases of claims for guarantee. B gets no further information about A.

2. A pays the DVD player by installments. Both A and B sign a contract for a credit and A has to prove his identity to B. Both parties get their own copy of the contract.

3. A buys 10.000 DVD players from dealer B by credit. If A and B have never met each other before, a signature will not be a sufficient prove of A's identity. They go to a notary who also signs the contract in order to certify it. A, B and the notary get a copy of the contract.

If there is a dispute between A and B, both sides can present their copy of the contract. In case 3 it can happen that one or both parties have manipulated the contract or even claim that they have never seen it before. Together with the copy of the notary an independent court can determine the truth and adjudicate correspondingly.

It is the aim of non-repudiation to transfer the concept of proofable *contracts* to user actions in IT systems, especially *electronic* judgment and enforcement are important issues. A non-repudiation service in a distributed computing environment does only make sense when at least the following requirements are fulfilled:

- All participants have trust in the concept and the functionality of the service.

- All participants will accept the results of a validation of proofs.

- Judgments must be enforceable, even against the will of a party.

- A legal binding has to be established for the basic cryptographic mechanisms that are applied in order to achieve the functionality of non-repudiation (e.g. digital signatures).

The basic concept of a non-repudiation service in a distributed system is that it offers an interface in order to make entities responsible for their actions: an irrefutable proof will be constructed for important events, e.g. the receipt of a message. A validation of this proof can be performed immediately (or at a later date) by independent entities of the security architecture.

A proof contains information about the event and some notion of non-repudiation policy that has been applied during event generation. Important parameters of the event are the event source, the issuer of the proof and a corresponding time stamp. The non-repudiation policy has to specify the underlying mechanism for the generation of proofs (e.g. a specific protocol for digital signatures), trusted third parties and trust relationships in the distributed system. The entities for the validation of a proof determine the origin and the integrity of the information.

In [6], [7] and [8] ISO defines non-repudiation as a part of the ISO/OSI security architecture. Note that the ISO framework does not specify what the consequences for *lying* are, because this would be too application specific. The OSI non-repudiation model is related to the *events* of sending or receiving a message. There are different types of non-repudiation for both types of events:

- Proof of origin: The recipient of a message receives a piece of evidence that proofs that the sender has sent exactly this message with its specific content to the receiver. Typically the evidence is sent together with the corresponding message.

- Proof of receipt: The sender of a message receives a piece of evidence that proofs that the recipient has received exactly this message with its specific content.

Note that a *proof of delivery* and a *proof of submission* are not part of the ISO model. The first one shows that a message has been sent successfully, the second one indicates that a message has really been delivered to the recipient.

The usage of a non-repudiation service can be described in four phases:

1. Evidence generation,

2. Transmission, storage and retrieval of evidence,

3. Validation/check of evidence and

4. Settlement of conflicts.

5.1. External dependencies

A distributed non-repudiation service cannot be viewed in isolation. Depending on the applied mechanism, additional services have to be integrated in order to guarantee security for the non-repudiation service itself:

- Public Key Infrastructure (PKI): When digital signatures are used as the mechanism for providing non-repudiation, a PKI is required for key management.
- Secure Time Service: If the non-repudiation scheme makes use of time stamps, a trusted time service is required.
- Secure Storage: For the storage of electronic evidence a secure storage is needed. This can be done with a secure persistency service or a secure database.

The communication channel between the external services and the non-repdudiation service has to be protected in an appropriate way, i.e. integrity, confidentiality, etc. has to be guaranteed. For this purpose SSL or Kerberos can be used.

5.2. Non-repudiation vs. auditing

At a glance auditing and non-repudiation seem to have a similar functionality. A closer look reveals quite different purposes and significant differences that are presented briefly now:

- Audits should detect and avoid violations of security policies, whereas non-repudiation is applied in order to prevent subsequent denials of user actions.
- Ideally auditing should react immediately, e.g. by locking a user account. Non-repudiation is introduced for the subsequent settlement of conflicts between two parties.
- Auditing may record and evaluate many events in order to detect an attack, whereas non-repudiation generates pieces of evidence in order to settle conflicts.
- Usually events are audited on a central audit facility which checks them periodically, non-repudiation does not perform periodical checks.

We therefore think that it makes sense to have distinct services for auditing and non-repudiation. A combination of both services is imaginable in certain scenarios, e.g., non-repudiation can protect the audit module against manipulated events.

```
public interface BillableLease extends Lease {
  Contract getContract() throws RemoteException,
                            NoSuchContractException;
  long getCost() throws RemoteException;
  Evidence renew(long duration,
                 Evidence proofOfOrigin) throws
    LeaseDeniedException, UnknownLeaseException,
    IllegalEvidenceException, RemoteException;
  Evidence cancel(Evidence proofOfOrigin) throws
    UnknownLeaseException,
    IllegalEvidenceException, RemoteException;
}
```

Figure 2. The `BillableLease` interface

6. Proposed architecture

Our architecture can be divided into two parts: a number of services available at run-time and a set of rules that services have to obey. First, we define the rules a service that charges for being accessed has to obey. Second, we describe the services needed at run-time. We then show how all participants interact.

6.1. Rules for services

As already mentioned, it is necessary to be able to link a lease to a specific contract. In our architecture, when starting to use a service, an identifier of the contract—in our case a reference to a `Contract` object—has to be supplied as a parameter. Of course, every `BillableLease` object (interface shown in Figure 2) can be asked for the contract its charging and service access policy is based on. Contracts must therefore be remotely accessible objects. We envision the use of signed XML documents as contracts. To enable the use of any kind of contracting, we are modeling contracts as objects.

```
public interface AccessCharge {
  AccessGrant start(Contract termsAndConditions,
                    Evidence proofOfOrigin,
                    long duration)
    throws NoSuchContractException,
           IllegalEvidenceException,
           RemoteException;
}
```

Figure 3. The `AccessCharge` interface

To be able to identify services with associated costs, we use the following method. Services implement the well-known interface of the service that they provide—just as they would if they would not charge anything. In addition to this, every such service must implement a special Java interface. This interface is shown in Figure 3 and it only contains one function which takes a `Contract` and a proof of origin and returns an object which includes a proof of receipt and a `BillableLease` which governs the service access time. Before starting to use a service and after lease expiration or cancellation, service usage must be blocked. All functions of the service should therefore be able to generate a `RemoteException`.

6.2. Required services

For making billable leases as easy to implement as possible and for reducing redundancy, we describe a number of services that have to be offered by the infrastructure. The services are regular Jini services and therefore register with local lookup services. Other entities can easily find these services by asking the lookup service for services implementing the appropriate interfaces. Our architecture is based on the existence of the following services: a non-repudiation service, a time service, and an accounting service.

Non-repudiation service. The non-repudiation service is a main component of our architecture and fulfills the following roles:

- it creates pieces of evidence and
- it persistently stores the evidence.

As is true for all Jini services, the interface of the non-repudiation service is well-known, but its realization is not. The deployment of different technologies for generating proofs is therefore possible and migrating to a different technology is as easy as starting the new non-repudiation service and terminating the old instance. In principle, the roles of the non-repudiation service can be achieved by using different service interfaces for each role. For the sake of simplicity we think of one single service interface in this paper.

In addition to generating proofs and returning them to clients, the non-repudiation service stores them in persistent secure storage. The proofs can later be retrieved in case of disputes. Although this functionality could be realized by a separate storage service, we decided to combine these two functionalities for performance and ease of implementation.

The function of an arbitrator does not necessarily need to be part of our architecture. A dispute usually arises after the service user received a bill, i.e. human interaction is required. The bill together with the evidences of both sides can be used as input to an external arbitrator service, too.

Note that both client and service provider must have a common understanding of the format of the non-repudiation proof. Depending on the contract between them, one or more pieces of evidence may be necessary to proof certain events. At least the following pieces of evidence for the described Jini scenario are needed:

- The contract that has been applied.
- The type of action or event, e.g., the method name that has been called.
- The parameters of the method.
- A digitally signed time stamp.
- The identity of the issuer of the proof.
- The identity of both parties, i.e., the client as well as the service provider. This can be done with a digital signature.
- Hash values for the validation of the integrity of the proof.

Time service. The time service is a trusted entity that supplies time stamps representing the current time to interested parties. Clocks on different computers in a distributed system usually do not have synchronized clocks. It is therefore mandatory to have a time-supplying service that both service and client trust. Time stamps are needed as leasing involves duration-restricted service access. Time stamps are acquired and stored as part of non-repudiation evidence by clients when requesting, renewing, or canceling service access and by servers when granting and terminating access to its service.

Accounting service. The accounting service is responsible for storing and retrieving billing records. In the easiest implementation, this service is a simple Jini-enabled front-end to a database. The database must contain at least the date and duration of the service usage, the type of service used (if there are more than just one kind), an identification of the client, and a reference to the contract which governed service access. Such an accounting service is still only usable in a rather restricted environment. If a service provider is offering a large number of different services, a more flexible and generic accounting service becomes necessary. It might be a good choice to allow arbitrary objects to be stored. On the downside, the billing application gets more complex, as it has to deal with a large variety of accounting objects.

An added functionality of an accounting service is that up-to-date billing information can be retrieved by clients. This functionality should be kept separate from the service provider's internal interface for storing billing records. Both functionalities should therefore be described in different interfaces. It might also be a good idea to register an individual object per interface to discourage fraudulent access.

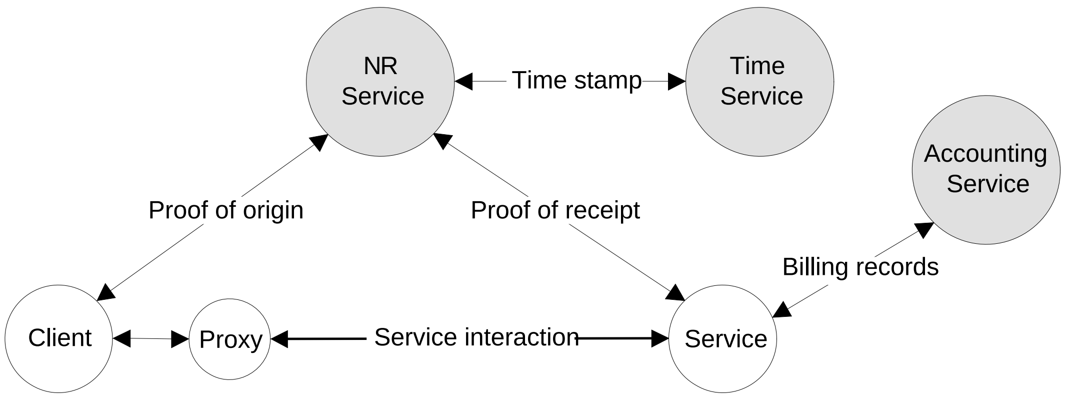

Figure 4. Architecture components

6.3. Component interaction

It is mandatory for all components to authenticate their communication partners. Equally important is the encryption of all communication channels to ensure privacy and integrity of exchanged data. While [5] describes a method to allow clients and services to authenticate lookup services and vice versa, it does not provide for authentication of services and clients to each other. This has to be implemented by all entities individually. An easy solution is to use SSL [1] which provides the required functionality. It is important to note that in a Jini federation communication is the sole responsibility of the service and its associated proxy. The client does neither implement communication functions nor can it enforce constraints on the type of communication going on between the proxy and the service provider. The service proxy must therefore offer functions to access SSL certificates. This requires basic trust to be established in the proxy which can be achieved by using the architecture described in [5]. In case only RMI is used for communication between entities the methods described in the RMI Security Extension [13] can be deployed. Appropriate constraints regarding authentication and encryption can be attached to proxies and individual functions.

The communication relationships between the individual components of our architecture are depicted in Figure 4. Of course, all components interact with one or more Jini lookup services as well, either for service registration or service lookup. These connections are left out for the sake of clarity.

An example of a leasing session is depicted in Figure 5. Function invocations in their relation to relative time are shown. Here, too, communication with lookup services is not shown for the sake of clarity. We assume that no contract has to be established before requesting a lease. The client therefore first acquires a proxy of the service it wants to access. In this example, the proxy is assumed to be a simple CORBA or RMI stub. Next, the client authenticates itself to the service and vice versa.

Once trust is established, the client initiates the shown sequence to start using the service. It first acquires a proof of origin from the non-repudiation service (called proof_O in the figure). This proof includes a time stamp which is retrieved from the time service. Once the client received the proof, it asks the service via its proxy to enable service access. Among other parameters, it includes the proof of origin in the function call. The service provider only grants access if such a piece of evidence is supplied, as it proofs that the client requested service access. The evidence might be later needed in case of dispute. The service provider now asks the non-repudiation service for a proof of receipt which it returns to the client (marked proof_R). The client now has a proof that the service agreed to supply its service to the client. Together with the proof, the service returns the billable lease which allows the client to ask for accumulated costs, the associated contract, and the expiry time.

When a lease is renewed or canceled, a similar sequence of function calls occurs. First, the client acquires a proof of origin, calls the appropriate function on the service provider and gets back a proof of receipt.

The access to the service is terminated either when the lease expires or when it is explicitly canceled by a function call of the client. As soon as this happens, a billing record is sent to the accounting service.

A non-repudiation of origin can be enforced as the recipient need not accept messages without a corresponding proof. The non-repudiation of receipt depends on the goodwill of the recipient. Without significant overhead, neither the sender nor a trusted third party can determine whether or not a message has been received successfully. Although this is not really fair, the sender (i.e. the client) has no financial loss if the service provider somehow does not create and return a proof of receipt. In this case he cannot use the service and certainly does not need to pay for it. In the case

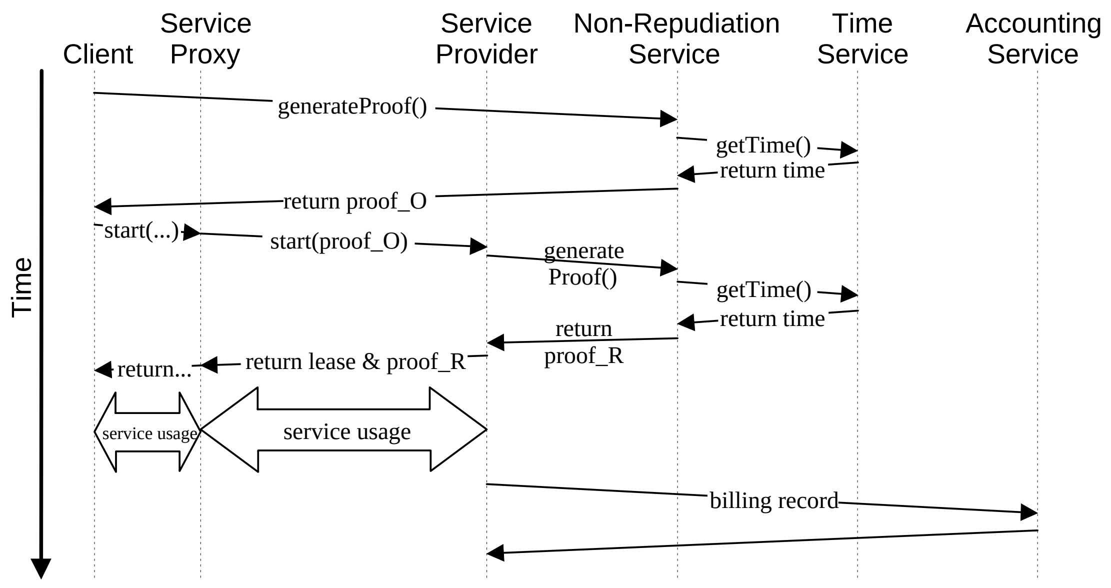

Figure 5. A leasing session

of the service provider not generating a proof of receipt, a dispute can be settled by asking the non-repudiation service for that specific piece of evidence. As it does not exist, the dispute is settled easily. In case the service provider generates the proof of receipt but does not pass it on to the client, the client will be blocked from any further service access. As mentioned before, this service access must be secured by non-repudiation methods as well. But because proving that a service was really provisioned is application-dependent, we do not consider this issue here. Besides this, creation and exchange of evidence should be wrapped in a transaction. The transaction only succeeds if both parties acknowledge the receipt of the other party's evidence.

7. Implementation

To demonstrate the viability of our architecture, we implemented a simple prototype. It consists of the proposed set of Jini services, namely a non-repudiation service and an accounting service. For ease of implementation we dropped the trusted time service and put the time stamping functionality into the non-repudiation service. We also refrained from implementing persistency of proofs and billing records. Figure 6 shows a graphical representation of the Jini federation while all required services are running.

Our non-repudiation service generates pieces of evidence by signing the data coming from a client or a service plus a time stamp and issuer information with its private key. A piece of evidence contains the data described in Section 6.2. The validity of a proof can be verified with the corresponding public key of the issuing non-repudiation service.

The billable leasing infrastructure was built on top of the Landlord protocol which is used in Sun's reference implementation of Jini. To test our architecture, we also developed a storage service and an accompanying client. The storage service's functionality is to store and retrieve arbitrary named objects. Of course, the use of that service is only possible while a valid lease exists.

Restricting service access to times when a valid lease exists requires identifying clients upon each service access (i.e. function invocation). As the basic service interface does not have any method to identify a client and as it was our intention not to change that interface, a different approach was chosen. The storage service does not send a

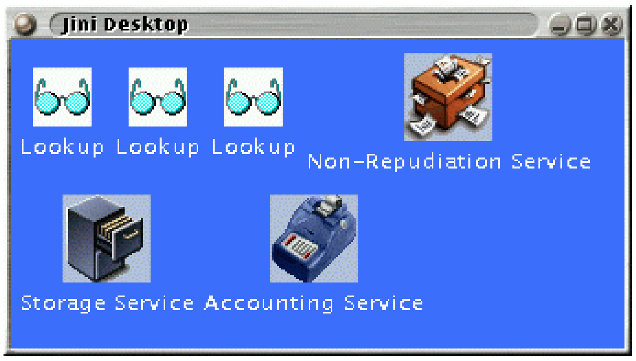

Figure 6. Our prototype in action

simple RMI proxy to its clients, it rather sends an "intelligent proxy". The main functionality of that proxy is to attach a unique identifier to every client. The identifier is sent to the proxy when service access starts, i.e., together with the `BillableLease`. The proxy now sends that identifier back to the service provider together with every function invocation. Using the identifier, the service provider can find the lease given to the client and make sure that service access is currently allowed.

8. Related work

The OMG added the CORBA Security Service [11] to the CORBAservices specification in order to counter common threats in distributed systems. As written in [9] it could *provide a high level of security for shared information and shared applications in distributed environments*. Unfortunately the specification of the CORBA Security Service includes ambiguities and gaps. The specification of the CORBA non-repudiation service, which is declared as an optional service, describes the functionality of such a security service but lacks interface details and interoperability. The good news is that the OMG has already realized these insufficiencies and works on modifications and the integration of missing concepts, e.g., CORBA PKI or CORBA firewalls. Note that CORBA Security is not a lightweight security component by nature, i.e., we do not think it is suitable for "thin" electronic commerce applications.

In the EU-funded project MultiPLECX [10] a generic non-repudiation service for CORBA has been implemented. Within this project it has been proposed to encode the non-repudiation tokens using digitally signed XML documents specified by the proposed Internet standard Digital Signatures for XML. The approach was to provide a separation between the application business logic and the generation of evidence allowing non-repudiation support to be incorporated into applications with a minimum of programmer effort. Thus the interfaces do not conform to the standard CORBA Security Service interfaces. More details can be found in [19].

Our architecture relies on the mutual authentication of components. As communication in a Jini federation is handled by service proxies, each service must implement its own authentication scheme, probably making use of a well-established system. Prior to performing authentication the client downloads a service provider's proxy. This is a piece of mobile code which might not be trusted by the client. In [5] we describe the design and implementation of an architecture that allows entities to establish basic trust in proxies and therefore in the components supplying the proxies. This builds a base for the architecture presented here.

This paper does not deal with how to establish, represent, or enforce contracts. [4] gives an overview of the requirements for electronic contracts as well as how contracts can be established and enforced. [12] uses XML to describe the terms and conditions of contracts (called "trading partner agreements" (TPAs)). As contracts are written in XML, they can be electronically processed and exchanged. This offers the possibility of automatically enforcing the terms of a contract. XML documents can automatically be transformed into objects representing the contract. Such an object could very well serve as a `Contract` object as we use it in this paper. Besides describing terms of the contract, the DTD described in that document offers the possibility of including security related information, such as certificates, in a contract.

Electronic cash was mentioned as one way to enable anonymous service access. But, depending on the scheme, e-cash itself might not be fully anonymous. [2] discusses a number of issues related to the anonymity of possible e-cash incarnations.

9. Conclusion and future work

In this paper we discussed an architecture for secure billing of leased services. We identified a set of services that are required to support this functionality and ease implementing services as well as clients. In addition, we looked at the interaction of the components and found some drawbacks, like the unfairness of the non-repudiation protocol. We think that this is acceptable, but further investigation is required. Although non-repudiation is an important issue for electronic market places, a look at the related work revealed that there is no usable and standardized non-repudiation service, especially for spontaneous environments like Jini.

As we focused on the architecture, there is still a lot of work to do. Some of the next steps are to integrate the described services with an external security infrastructure, e.g., SSL for a secure communication channel. As we rely on public key cryptography, Jini has to be tied to a PKI. An interesting point will be to introduce facilities for the negotiation of contracts. Besides that the anonymity of clients and service providers has to be analyzed in detail.

But even if all these problems are solved, there is still an even bigger question to be answered: is it possible to built a dynamic, spontaneously networked system that is sufficiently secure? Jini enables dynamic interaction between components in a distributed system. Although our components do interact dynamically, each one of them has to have a rather large set of a priori knowledge, especially the identities of all other components it communicates with. Therefore, a large part of the spontaneity of Jini is sacrificed to security. The challenge is to find a more favorable trade-off.

References

[1] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol, Version 3.0. Internet Draft, Mar. 1996.

[2] P. S. Gemmell. Traceable e-cash. *IEEE Spectrum*, 34(2):35–37, Feb. 1997.

[3] C. G. Gray and D. R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 202–210, Dec. 1989.

[4] F. Griffel, M. Boger, H. Weinreich, W. Lamersdorf, and M. Merz. Electronic Contracting with COSMOS—How to Establish, Negotiate and Execute Electronic Contracts on the Internet. In Z. M. C. Kobryn, C. Atkinson, editor, *2nd Int. Enterprise Distributed Object Computing Workshop (EDOC '98)*. IEEE Publishing, Nov. 1998.

[5] P. Hasselmeyer, R. Kehr, and M. Voß. Trade-offs in a Secure Jini Service Architecture. In *Trends towards a Universal Service Market (USM 2000)*, Sept. 2000.

[6] ISO. ISO 7498-2: Information Processing Systems - Open Systems Interconnection - Basic Reference Model - Part 2: Security Architecture, 1989.

[7] ISO. ISO 10181-4: Information Technology - Security Frameworks for Open Systems: Non-Repudiation Framework, 04 1997.

[8] ISO. ISO 13888-1: Information Technology: Security Techniques - Non-Repudiation - Part 1: General, 12 1997.

[9] U. Lang. Security Aspects of CORBA. Master's thesis, Royal Holloway University of London, 1997.

[10] MultiPLECX. Multi-party Processes for Large-scale Electronic Commerce Transactions. http://www.multiplecx.org, 1999. EU funded ESPRIT project.

[11] OMG. Corba Security Specification. http://www.omg.org, 1998.

[12] M. Sachs and J. Ibbotson. Electronic Trading-Partner Agreement for E-Commerce (tpaML). `http://www.xml.org/tpaml/tpaspec.pdf`, Jan. 2000.

[13] Sun Microsystems Inc. *Java Remote Method Invocation Security Extension (Early Look Draft 2)*, Sept. 1999.

[14] Sun Microsystems Inc. *Jini Architecure Specification – Revision 1.0.1*, Nov. 1999.

[15] Sun Microsystems Inc. *Jini Distributed Leasing Specification – Revision 1.0.1*, Nov. 1999.

[16] Sun Microsystems Inc. *Jini Lookup Service Specification – Revision 1.0.1*, Nov. 1999.

[17] Sun Microsystems Inc. Jini Connection Technology Vision Scenarios. `http://www.sun.com/jini/vision/scenarios.html`, 1999.

[18] J. Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[19] M. Wichert, D. Ingham, and S. Caughy. Non-repudiation Evidence Generation for CORBA using XML. In *15th Annual Computer Security Applications Conference*, pages 320–327. IEEE Computer Society, Dec. 1999.