CORE

Version 2.0

Table of contents

Table of contents....2
Foreword....4
Introduction....5
Client-server architectures....5
The Jini Networking Technology....8
The FETISH network....16
FADA enters the picture....17
FADA in depth....19
FADA Topology....22
FADA behavior....26
Service Directory Architecture....28
Graphical overview....31
Diagram description....32
FADA communications layer....35
RMI and the new FADA communications layer....38
FADA mechanisms....41
Discovery mechanism....41
Registration mechanism....43
Lease expiration mechanism....46
Lease renewal mechanism....46
Lease cancellation mechanism....53
Lookup mechanism....53
Retrieval of proxies....59
The FADA stub - skeleton compiler....62
Use of different communication protocols....64
Multicast extensions....69
Multicast events....69
Events hierarchy....70
Multicast discovery....71
Multicast Announcement....72
Registration events....73
Code Examples....75
Discovering nodes using multicast discovery....75
Registering as RegistrationEvent listener....76
Registering a Service....78
How does the net.fada.directory.SignedMarshalledObject work?....81
Deregistering a service....82
Looking up a Service....83
How does the matching mechanism work?....85
Complete example....85
Service provider side....86
Client side....91
Notes about JAVA sandbox and dynamic code loading....92
The FADA stub - skeleton compiler: an example....98
Easier development of the server: use of fadagen....98
Skeleton/stub and proxies....102

FADA security wrappers.....109

Security background.....110
Computer Security.....110
Decentralized trust management.....110
Java Security Architecture.....111
Java language and platform security.....112
Resource access control.....113

Requirements for FADA Security.....115
FADA network security functionality.....115
FADA's users security functionality.....116

FADA and Federations.....117

Dynamic and secure loading of remote code in FADA 119
SignedMarshalledObject.....119
SignedMarshalledObject's structure.....119
Step by step: Secure instantiation of a remote object in FADA.....121
SignedMarshalledObject instantiation.....124
Helper classes for certificates and keys manipulation.....127

Default FADA node security wrapper implementation 129
Secure communications between Fada node and its proxy.....129
Notes about policy files configuration.....132
Java policy files.....132
Fada policy file.....133

Velocity and FADA Web based management.....135
Templates and variables.....135
Web based management: Themes.....138

Fada Plugin Architecture.....140

Installation and Setup.....141

Administration of the FADA.....145

Appendix A.....148
A example of FADA policy file.....148

Bibliography.....150
Graph algorithms.....150
Jini over Internet.....150
The Kalman filter.....150
Network delays and algorithms.....151
Jini.....151
RMI.....151
Java.....151
Java based technologies.....151
Other.....151
Security background.....153

Foreword

This is the complete FADA manual. It includes all information needed to understand what FADA is, what it is designed for, how it works, and how to install it and use it.

The FADA started as a series of patches to enable the use of the Jini Networking Technology on the Internet. Later on, it was decided to completely drop the Jini reference implementation, and start a work on its own, because the size of patches started growing so much that it added considerable weight to the already heavy Jini reference implementation.

In its initial stages it was seen that, no matter how many patches were applied, FADA was lacking control over the execution of the Jini reference implementation, so parts of it were rewritten from scratch. A few internal releases later it was realized that there was so little left of the Jini reference implementation it was no use to keep using it. That day, the FADA entered its adulthood.

Along with the primary use of the FADA, a set of other technologies that surrounded the concept of FADA materialized as tools and utilities useful for the service provider programmer. Some of them have been discarded, but the rest have been included in this release of the document *Core FADA*, along with some examples of its use.

We hope you find the information contained here complete and useful.

Enjoy it.

The FADA development team.

Introduction

Networking software has been a little around, and some drawbacks and intrinsic difficulties have been identified in this time. With the explosion of the Internet, favored mainly by the appearance of the World-Wide-Web in 1990, the network has become the arena of new business. This trend has pushed the development of new or not-that-new networking technologies.

One of the projects that have appeared in the new information society is the FETISH project. The FETISH project is an open distributed platform for the exchanging of services and applications, which has been developed through an European Commission IST Frame Program 5th funded project using the SUN JINI Network™ Technology. In words of the FETISH project officer, “our mission is to turn the scattered tourism enterprises, using disparate systems and platforms, into a seamless network available from a common access point”.

Throughout the development of the FETISH project, one of its modules, called the FADA, has become an entity on its own. The FADA is the Federated Autonomus Directory Architecture. Federated, because it is an architecture composed of several computing entities referred to as FADA nodes, that work together to perform the system’s functionality. Directory, because it acts as a directory or repository of service proxies. Autonomous because it is not necessary human administration for evolving the network created by the nodes interconnected. Architecture because the term FADA doesn’t only refer to a piece of software, but to a whole system of software entities.

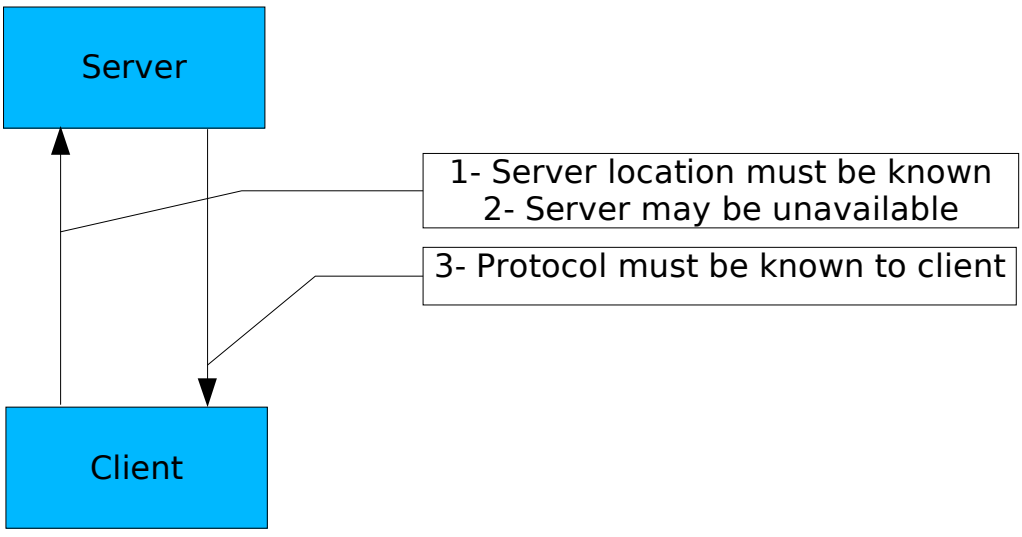
Client-server architectures

Let’s start with the basics. A common client-server application is composed of two entities, named client and server. The client is a software entity (a piece of software) that requests some functionality (by a human user, or by yet another client) that it can’t provide on its own (for lack of computing resources, or lack of secondary storage, or whatever reasons). Common client-server architectures widely used and known are, for example, the Web, in which a program, the web browser acts as the client, requesting contents in the form of HTML documents to web servers, that play the role of the server.

Other examples include the relationship between a database management system and the consumers (and providers) of data from the database. The DNS service is also a client-server architecture, in which the client wants to know the mapping between a name and the corresponding registered IP address. The DNS server contains that information, so the client simply fills in a request that is then sent to the DNS server, which performs whatever computing is needed and returns a response, in a form understandable by the client.

Client-server architectures have many pros. One of them is that, by decoupling the computation in two entities, these two roles need not to be executing in the same machine, so clients can be computationally poorer machines than servers. Another pro is that a server may accept requests from more than one client, thus maximizing resources usage.

However, client-server architectures also have some cons. First, if the server is not working (the machine that hosts it is down, or the software is broken) the clients can not perform the required computation, no matter if the clients' machines are perfectly well. Another con is that if the server and the client reside in different



machines another point of failure is introduced: the network.

An annoying feature of client-server architectures hosted on different machines is that, if the server fails, there's no way for clients to know it until they perform the request. Depending on the type of service and/or failure of the server, the client will get an error, or the connection will time out. Only after time has been wasted on part of

the client it will realize that the computation can not be completed by that server, and so it is up to the client (the software or the human behind) to decide whether to give up or try with another server (if such another server exists). Wouldn't it be nice to know in advance whether the server is working without the need to perform the request and watch it fail?

Another characteristic of client-server architectures is that the client must know in advance how to contact the server. The service may (or may not) provide a method to find the location (in the network) of the server. That's one of the main reasons for the proliferation of portals and search engines for the web: to find the host that contains the information needed.

What's more, the client must implement the protocol understood by the server, so multitude of servers imply there are multitude of client implementations. Even servers that provide similar functionality may use different protocols for communication with clients, so two clients that would otherwise be virtually identical with respect to the calculation performed differ totally with respect to the protocol implemented.

To solve some of these problems, in 1998 Sun Microsystems Labs developed a networking technology that eliminates or otherwise alleviates some of these drawbacks.

The Jini Networking Technology.

Jini(TM) network technology is an open architecture that enables developers to create network-centric services -- whether implemented in hardware or software -- that are highly adaptive to change. Jini technology can be used to build adaptive networks that are scalable, evolvable and flexible as typically required in dynamic computing environments.

The Jini Networking Technology takes the whole problem of client-server based applications, makes some basic assumptions, creates some abstractions, and provides solutions for the aforementioned problems of client-server architectures.

Some of the basic assumptions are:

- The client is interested in the service provided by the server, not in the protocol involved to perform requests. That's an unavoidable, but secondary, item.

- The client would like to be agnostic as to what server it needs to contact in order to perform the request. All it knows is the type of service. All it wants to know is the type of service. The name or location of the server that provides the service is an implementation detail the client would be quite happy to be completely unaware of.

- The client wants to know in advance if a certain service is available. That requirement is twofold: the client wants to know what services are available, and the client wants to know what servers not to contact, because they are not working. Put in other words, the clients wants the list of only working servers.

- The client doesn't want to be forced to know the implementation details of the protocol for the communication with the server. Or, if that is not possible, the client would like the protocol to be as simple and consistent as possible. The protocol used by the client should be related to the business logic performed by the client, and not to the implementation details needed by the communications protocol.

- The solutions should work with existing working services without modification, or at least keeping those modifications to a minimum.

The Jini Networking Technology was made possible thanks to the existence of the Java programming language. In fact, Jini is completely based around Java. This, however, doesn't mean that the servers must be implemented in Java at all. However, clients must have a minimum knowledge of Java in order to interact with Jini.

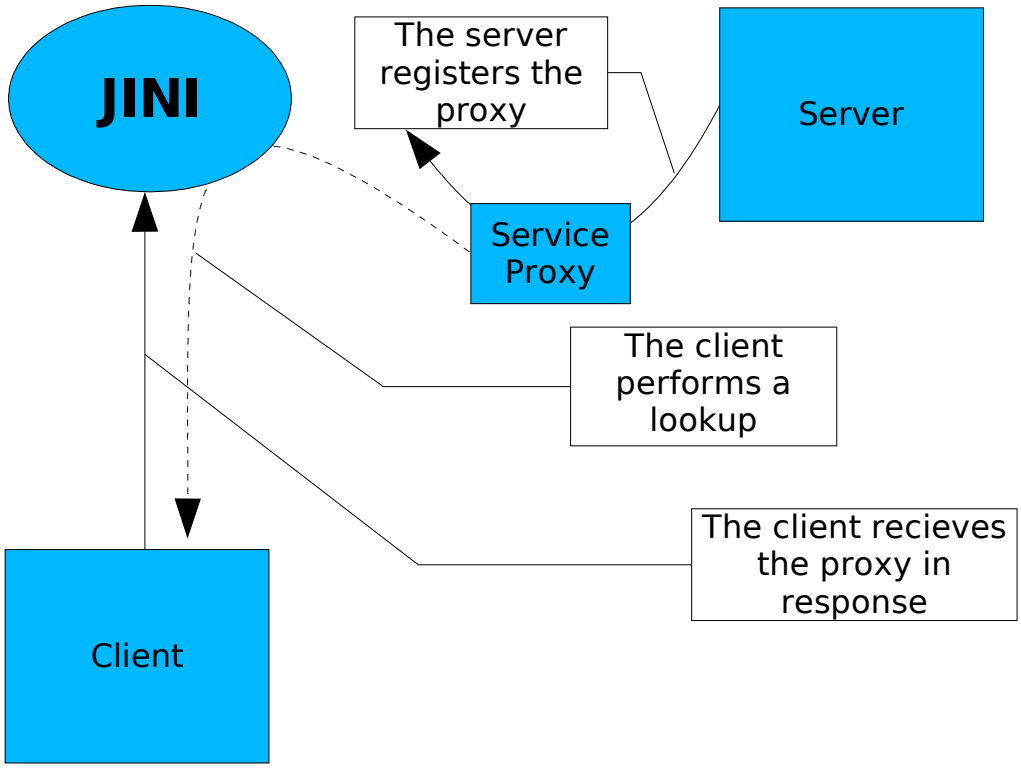
Java provides some functionalities that made possible the appearance of Jini. The related functionalities are, mainly, the following:

- Java programs (called *classes*) execute in a *platform independent virtual machine*. That is, the same Java class executes well on different hardware platforms, with different operating environments, provided that an implementation of the Java Virtual Machine (JVM from now on) is available for a particular hardware-operating environment combination.

- Java classes are able to be converted to/from a stream of bytes. This stream of bytes can travel across the network and be reconstituted in other place (another JVM). This portability of classes affects both the class data and the class code.

- Java provides an easy mechanism to define the operations performed by a class, even when that class is completely unknown to the user of that class. That is, if the user of a class knows the names of the methods that class provides, it doesn't matter where that class has come from: the user of the class will be able to call such methods. Java does so by defining `interfaces`, that is, classes with just the methods' names, parameters and return types, with no implementation. If a class `implements` a certain interface then it is guaranteed that those methods will do something when invoked upon the class. That means that, if a user gets an instance of a class it doesn't know, but he is assured the class implements an interface he knows, he can call the interface methods on the class, and the class methods will be executed just as if he had known the class from the beginning.

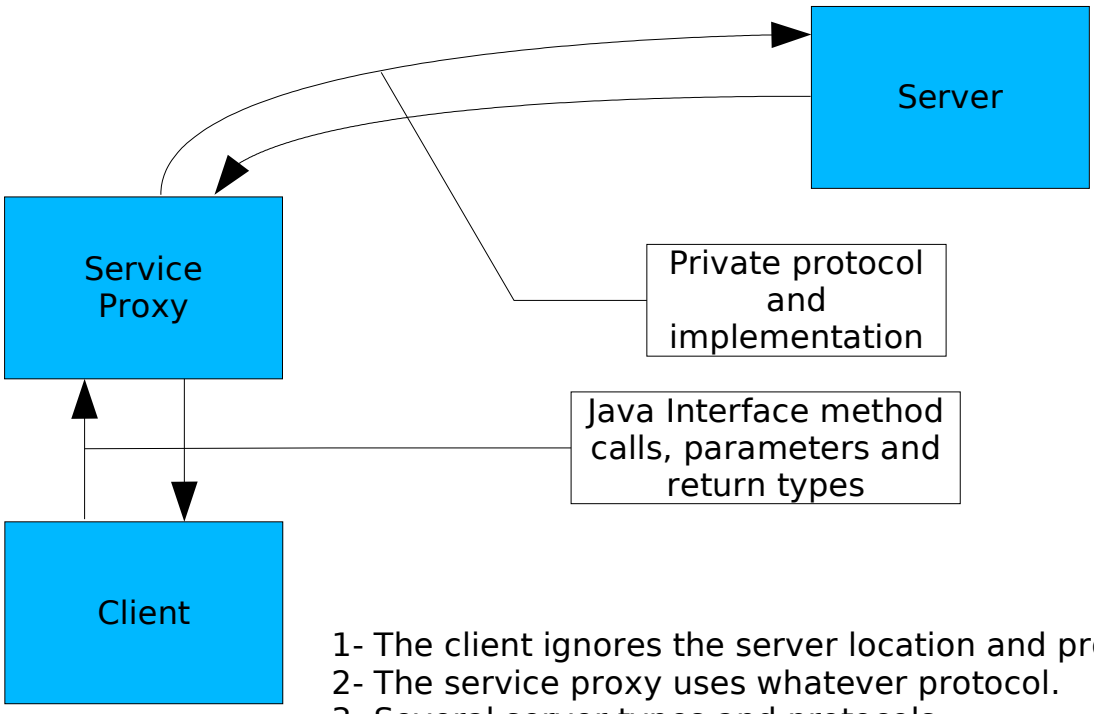
Jini defines a service as a Java interface. That is, it translates requests to the server to invocation of methods of the interface.



Jini also decouples the interaction between the client and the server by interposing an agent, called the `service proxy`. This service *proxy* is a Java class that implements a certain service interface, known to the client, and therefore the client is able to invoke its methods, although the client doesn't know implementation details of the proxy, such as the class name.

The service proxy methods perform the actual request to the server. In this way, the protocol details are hidden among the implementation details of the proxy, thus isolating the client from them.

To provide the client with a fresh list of working services, Jini provides a central repository (many independent instances of which may be present in a network) that may be polled by using some Jini utility Java classes. These classes make no mention at all about the actual location of the central repository or the servers that provide the actual services, so the client is completely isolated from those details.



1- The client ignores the server location and protocol.
2- The service proxy uses whatever protocol.
3- Several server types and protocols may implement the same service interface.

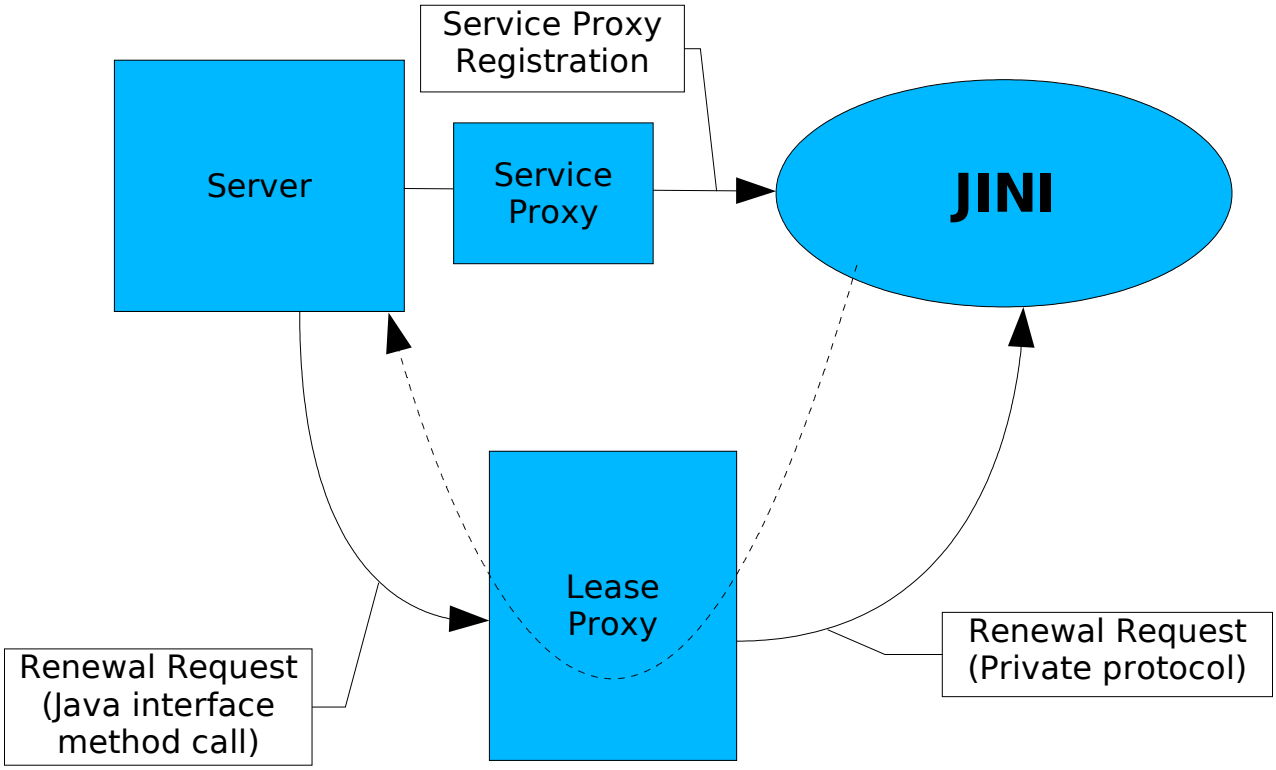
The service proxies that interact with the servers are known to the Jini central repository (called lookup server, LUS from now on) because the service providers have registered them in the LUS. The registration involves the creation of the service proxy by the service provider. Once the service proxy has been created, the instance is converted to a stream of bytes and sent to the central repository, along with some identification information.

These registrations, though, are not permanent. By giving another turn to the screw, the designers of Jini decided here to avoid another of the main annoyances of client-server architectures: non-working servers.

When a service provider registers a service proxy in a LUS the latter provides the former with an entity called `lease`. The lease is an entity that represents the compromise of the service provider to notify the Jini LUS that the server is working. The LUS grants the service provider that the service proxy will be kept in the LUS for a

certain amount of time. After that time, the service proxy will be silently discarded from the LUS, unless the service provider `renews` the lease for an extend amount of time. The lease renewal procedure is easily performed by the service provider, because the lease is another Java class that contains the suitable method.

That is, the service provider registers a service proxy in the LUS and gets in return a Java class that implements the standard Jini interface `Lease`. The actual class returned by the LUS is of little interest for the service provider, because all he wants to do is invoke the methods defined in the `Lease` interface. Yes, the actual class returned by the LUS is a *proxy* for the lease.



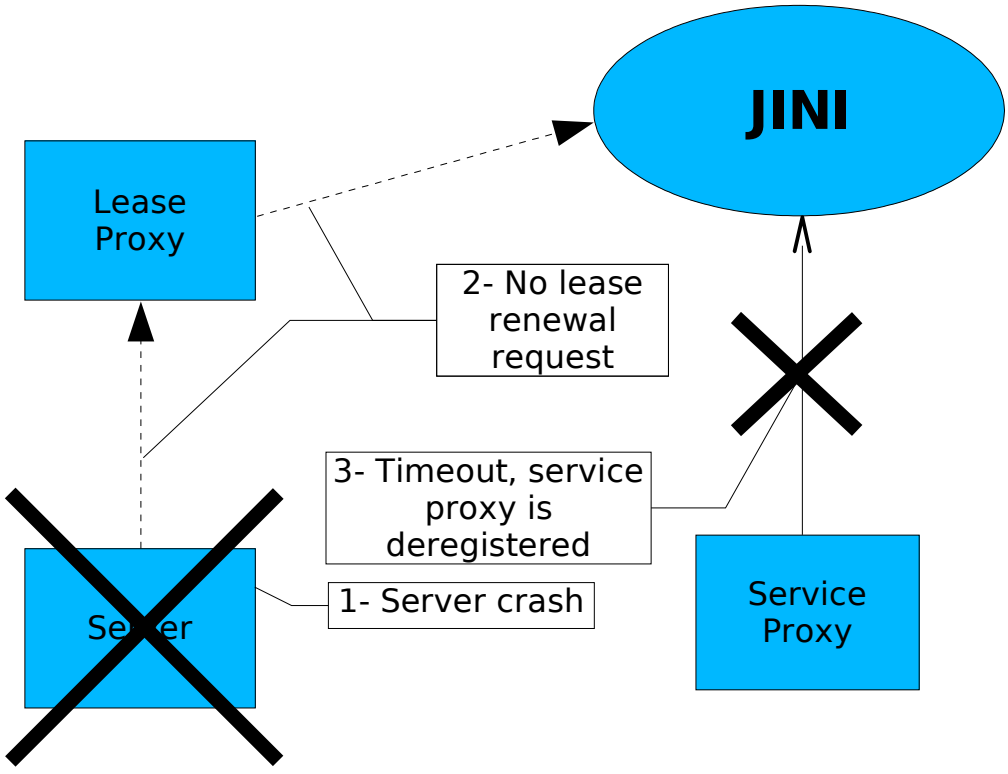
The Lease interface has methods to perform the renewal of the lease, and thus extending the life period of the registration of the associated service proxy in the LUS. It also provides methods to cancel that lease, thus notifying the LUS that the associated service proxy must be discarded and deleted from the LUS internal registry.

The renewal of lease must be done by the service server itself, whenever possible. In this way, should the server crash (and therefore be unable to notify anyone about its own decease) the

lease would eventually expire, and the service proxy for the now deceased server would be deleted, without requiring any human intervention to maintain the LUS internal registry.

The lease can also be canceled (by calling the appropriate methods) if the service knows is about to be stopped. In this way no clients will get the service proxy for the service that is about to stop functioning.

In cases where the lease can not be renewed by the server itself (because the server is not written in Java, for example) it is possible to construct a wrapper in Java that polls the server about its state, or uses whatever means to know if the server is working. Should this wrapper detect that the server is not working anymore it should cancel the lease inmediately. If both the server and the wrapper are executing in the same machine, the situation is still better, because even a hardware failure that stops both the service and the wrapper will cause the lease to expire eventually.



Evidently, the higher the lease time, the higher the time a non-working service will be available to clients that request its service proxy, so the idea is to keep lease times as small as possible. Note

also that small lease times require a more frequent renewal of the lease.

Another side benefit of the Jini technology is that it enables the interaction of services with the absence of human intervention. As computer software is able to discover network services there's no need for a human agent to initially configure a server, or to reconfigure it after a failure of one of its components.

Well, that's roughly how Jini accomplished its goals. What has FADA to do with this all?

Although Jini serves its purpose, it still has some drawbacks that rendered it unusable for Wide Area Networks (WAN), such as the Internet. The main drawbacks are:

- Jini was primarily designed to work in a Local Area Network (LAN). Most LANs support some sort of broadcast or multicast, a mechanism in which one participant of the network is able to send a single message that is received by all (in case of a broadcast) or a group (in case of a multicast) of the other participants in the LAN. This is an efficient means of communication with a large group of machines. However, multicast is almost never usable on the Internet.

- The multicast mechanism is heavily used in Jini, and so most of the Jini procedures are impossible to perform in the Internet.

- The network delays of a LAN are typically in the order of milliseconds, or tenths of milliseconds. In the Internet, the network delays are typically in the order of seconds or tenths of seconds, a 1000:1 ratio.

- Some mechanisms and policies used in Jini rely on the fact that network delays are in the order of milliseconds or tenths of milliseconds. Therefore, those mechanisms and policies may not be used in the Internet.

- Jini also relies on the fact that every computer in the LAN can reach every other computer, or at least that all computers in the LAN that want to participate in the Jini Networking Technology can reach one another. However, in the internet this is not always true (and most of the time decidedly untrue), because of the *firewalls*.

- A firewall put around a network that is attached to the Internet may allow only some kind of network traffic to travel on certain ports of certain computers in the LAN. However, Jini needs arbitrary access to arbitrary ports in arbitrary hosts. This is because of the way in which the Java Remote Method Invocation (RMI) mechanism works. Jini relies heavily on RMI.

Jini allows the presence of various LUS in the same LAN. However, those LUSes don't cooperate to provide a more flexible functionality. It is up to the client of Jini to discover all those services and use them as independent registries, therefore overloading the client.

These considerations conclude that it is not possible to use Jini as it comes in an Internet environment. However, that's exactly what FETISH needed.

The FETISH network

The FETISH network is an open distributed platform for the exchanging of services and applications, which has developed through an EC funded project (IST-1999-13015) based on the Sun JINI Network(TM) Technology.

At present, the FETISH platform is being tested in real commercial environments.

"Our mission is to turn the scattered tourism enterprises, using disparate systems and platforms, into a seamless network available from a common access point". Following the tourism business challenges, the platform is a constant changing environment, allowing inter-operability and using Open Source technology for service connections through wired and wireless devices.

The FETISH network provides the tourism enterprises with a range of competitive advantages, such as:

- Electronic distribution efficiency.
- Faster deployment and development of new technologies.
- Lower IT costs.
- High-value applications availability without maintenance and integration tasks.

It is evident that Jini, as it comes, can’t be readily used for the FETISH purposes.

FADA enters the picture

Jini, as it comes in the reference implementation available from Sun, is not suitable for systems that must cross firewalls. It requires a quirky network setup and a not-so-obvious parameter setup on the client side. Tests performed on such environment showed poor reliability and performance. It was decided to drop the reference implementation and start software development from scratch. However, the concepts highlighted by the Jini Networking Technology are sound ones, so most of them have been kept, where applicable, and others have been extended in the development of the FADA.

So FADA was designed to overcome most of the limitations of Jini. Some of its key points are the following:

- FADA is an architecture of Lookup Servers that work together. Performing a lookup request on a FADA node will obtain the matching responses from that FADA node as well as from the neighboring FADA nodes, without any additional intervention of the client software.

- FADA allows the integration of services as well as Jini does, by providing similar mechanisms for service discovery.

- Each FADA node uses one and only one fixed port for communications, so it is easy for network administrators to select a free port and give it to the FADA node that will lie behind the firewall.

- FADA doesn't use Java RMI to perform remote communication, neither from the client to the FADA architecture, nor within the FADA architecture itself. Instead it uses a mechanism resembling the standard Java RMI, modified to work transparently over HTTP. Therefore, clients of the FADA can be in a firewalled network, as long as they have access to the Internet via HTTP, which is widely common.

- FADA makes no assumptions about the network delay. Instead it uses mathematical tools (Kalman filter) to model that delay, and adjusts itself to the properties of the environment it is working on. These tools are adaptive, so changes in the environment don't cause FADA nodes to fail.

- FADA doesn't rely on a broadcast medium to perform its operations. It can, therefore, be used in the Internet.

- FADA provides an HTML over HTTP view of its state. A simple web browser can be used to query the state of a FADA node, and even to administer it. There's no need for specialized separate tools.

- RMI classes mobility require that the class files that define a class behavior (its code) must be made available through a separate HTTP server. FADA relies on class mobility, but acts as its own HTTP server, therefore requiring no additional software.

FADA in depth

FADA stands for “Federated Advanced Directory Architecture”. It is a virtual Lookup Server in the sense that different Lookup Servers (FADA nodes) will work together to provide the LookupServer functionality from any entry point. Also, any of these Lookup Servers will cooperate with the rest to find implementations of services.

This is the major difference with Jini. With Jini you can have as many Lookup Servers as you want within a LAN, but the client is responsible to find them all (though facilities are provided). In Jini LookupServers don’t cooperate, so in order to find a service proxy the client has to ask them all. The good news is that broadcast protocols are fairly efficient in a LAN, so a client can make just one lookup request on all LookupServers. Unfortunately this can not be achieved in the Internet without additional logic and cooperation among LookupServers.

This is exactly what FADA provides. Furthermore, the FADA is a truly distributed system, in the sense that there is no central authority or common communication channel, and LookupServers inside the overall virtual LookupServer operation of FADA work in a peer-to-peer fashion. The overall topology is an hybrid centralized + decentralized topology, where the distributed FADA architecture acts as a centralized server from the clients’ point of view.

The FADA (as well as Jini) holds proxies for services. A proxy is a Java class that performs communication with a real service, and that is downloaded at run-time by clients. Clients use the public methods on the proxies to access the services. These public methods are specified in Java interfaces that service proxies implement. A FADA node is a service that acts as an entry point to the distributed database.

Service providers must implement Java classes that act as a gateway to use their services, which may be written in Java or not. In case they aren't, the Java class they provide to the FADA clients, the service proxy from now on, can use whatever method to communicate with the service. For example, the service could be written in C, and using JNI the service proxy could access the service. Or the service could be accessed through http, and the service proxy could open sockets to the proper ports to communicate with the service. Or it could be implementing some JXTA service. Writing a service proxy in Java doesn't force any kind of implementation on the

server side. Only a minimum of compatibility is requested. Within the FETISH framework service proxies follow the interfaces defined in the FETISH repository (a sort of UDDI-like database).

Note that these proxy objects will be executed in the client's machine, so this fact must be taken into account when designing and implementing service proxies.

We've seen the similarities between FADA and Jini. But, due the fact that FADA will work over whole Internet, FADA also has some differences from standard Jini:

- FADA nodes are designed and implemented to work together. This means that FADA nodes are ready to contact and collaborate with other FADA nodes by default. Although Jini Lookup Services can be federated, FADA has this behavior built-in, and users of FADA (both service providers and clients) don't have to deal with it.

- The Jini sample implementation was thought for working over broadcast capable, low-latency LANs, and uses some of the features of such networks to achieve its job. FADA is designed to work over the Internet. This introduces several differences:

  1. Jini uses broadcast to discover the available Jini Lookup Services; broadcast can not be achieved in the Internet: first of all routers refuse to propagate broadcast packets, but in case they didn't a single packet would flood the whole Internet to reach every single connected host. In case some of them responded with a broadcast too, this would provoke a packet storm, disturbing other hosts not involved in the communication. In a LAN this is not an issue. In the Internet this approach can not be used, so discovery is not performed (by now) by FADA nodes. They need an explicit reference to already existing FADA nodes. This problem is actually being worked on.

  2. Jini does some of its housekeeping thinking in the low delays of a LAN. For example, service proxy registrations are not granted forever, but kept for an amount of time called the lease time. Service providers get a lease object upon registration. If they wish to keep the registration up they must periodically renew this lease. Jini offers some ways to automate this mechanism for the service providers. But these mechanisms are based on the knowledge of a low delay for communication between hosts in the LAN. This

statement doesn't keep true for the Internet, where delays of several seconds are not rare. The default Jini behavior must be changed. FADA takes into account these high delays, and tries to adapt to them. Failure recovery in certain situations has also been contemplated and achieved.

3. Jini performs communication over a reliable network, i.e. a LAN. But FADA deals with the higher possibility of network failure of the Internet: the more involved parties, the higher the probability of error. It doesn't rely on reliable connections. This leads to an approach of doing things “maybe once”, instead of “at most once”, as it is not possible to assure things will be done exactly once. Despite that, FADA is not very sensitive to failures: only performance degrades.

Another great difference between Jini and the FADA is that the former uses RMI as the basic communication channel, while the FADA uses HTTP. Although RMI can be tunneled through http with the help of certain cgi script made available by SUN Microsystems, there is an implicit overhead when transferring objects through such channel. FADA nodes exchange potentially shorter messages to cooperate, and thus the overhead is diminished. Also, working with http directly allows clients to interact with the FADA through firewalls, provided that there is some kind of http proxy or tunnel available.

FADA Topology

Jini is designed to work in a network that is able to do broadcast. The announcement protocol used by the Jini Lookup Services to announce their presence to the whole community relies on a broadcast primitive offered by the underlying hardware/operating system. But in the internet there is no way we can assure a message will be received by every single connected host. Even in the ideal case that we could, every broadcast message would flood the whole Internet. That's a waste of bandwidth. A different approach is needed.

Instead of that, the FADA uses the DNS infrastructure, a well known domain name, and the ability of the FADA nodes to know their neighborhood (in terms of FADA nodes connected with them) to provide a way to discover new FADA nodes and their location. By location we mean we know its IP address and the port where they are listening for requests, so we know in which host it is really running.

Compare this with the Multicast Discovery Protocol used by Jini, in which a Jini Lookup Service is found by issuing a request to a well-known multicast IP address in a LAN. Every Jini Lookup Service that is running in a host connected to the LAN would receive the request (because all of them are listening on the same multicast address) and therefore respond. But a LAN offers methods to perform broadcast: in a tree or bus topology the medium is already a broadcast medium; in a token ring topology the message can be released by every repeater with low cost (only the ring latency, which is an already assumed cost); in a star topology it is also easy to perform a broadcast. In the Internet, however, this approach is not practical, as the vast majority of network routers block multicast traffic.

Jini also offers the possibility to use a Unicast Discovery Protocol, in which a normal (i.e. non-multicast) IP address is used to perform communication. But the underlying protocol and mechanisms are hidden from the programming interface. Although this gives easier use of the protocols and mechanisms, this approach allows little or no control at all, and therefore, if things don’t work as expected (as is the case with Jini in the Internet) there is little that can be done to solve the problem.

FADA builds a network of collaborating FADA nodes which use direct http connections to other nodes. It means that a given FADA node knows the addresses and ports of some other existing FADA nodes (called neighbors in the first degree or just neighbors). In this way, by using standard networking mechanisms instead of mechanisms

hidden under a higher level interface, it is possible to change the behavior to match the running scenario.

Several topologies for the FADA network have been thoroughly considered. The first one (which had been already implemented and tested) was a tree. This arrangement offers the advantage that there are no cycles, so it is easy to traverse the whole tree while performing searches. But it also has some drawbacks:

- FADA nodes can crash, thus isolating portions of the tree. A solution could be that every node had a list of alternative nodes to adopt as parents, so the structure would keep connected. But this leads to another problem: the root of the tree is a single point of failure. To avoid this, any of the root children could be elected to be root, adopting its siblings as children. But this tends to overload the root, and also a contention protocol to choose one of the children as root must be defined.

- Also, there must be a way to avoid cycles in the structure, as a given FADA node can connect to any other node. If a node is added to another one as a child, and then gets as a child the parent of its parent a cycle is created. This condition could be checked beforehand, but cycles can also exist by connecting to its grandparent, or its grandgrandparent, etc. It would be necessary to check the whole hierarchy before connecting, and this would take long time (think of a huge hierarchy with hundreds of nodes, and the links between them are running over TCP/IP). Furthermore, the hierarchy is not static, so while checking this condition any other node could be connecting already checked nodes, so the cycle would also appear.

In short, too many problems arised, and they seemed very hard, if not impossible, to overcome, so it was chosen not to force any structure in the FADA network. But it is possible to conduct the structure of the architecture by using the administration tools provided with the FADA Toolkit and the HTML administration interface contained in the FADA nodes themselves.

Any node can connect to any other node, creating transitive cycles. Direct cycles are avoided, as there is no reason to create one (relationships are already bidirectional). The FADA nodes topology is not fixed or known in advance. This gives greater flexibility to the federation: any node can, in any moment, join the architecture in any point, and automatically it will be reached by any client in the network that wishes so.

Also, a node can leave the federation in any moment: nobody will miss it (except, perhaps, the service providers that registered their services there; but they are free to register them in any other place: FADA nodes are idempotent). As a desired side effect, any given FADA node can crash, and the rest will continue to work without ever noticing it. Relationships between nodes take into account the possibility of failure, and non-working references are dropped without causing failure on the nodes. In short, administration has been kept to a minimum.

The fact that the FADA topology is neither known nor can be forced beforehand leads to a certain degree of performance degradation. But it is more important to ensure flexibility and reliability than sacrificing both for the sake of performance. Performance will degrade anyway because of the Internet congestion state. Put in other words: if the bottleneck is not in our hands, why keep a design (and implementation) whose efficiency will be hidden by the inefficiency of the network instead of a design (and implementation) whose efficiency is not very high, but works under bad conditions?

Several algorithms for efficient propagation of lookup requests (that are, from every point of view, broadcasts from one node to the rest) have been considered.

The first one was flooding, which consists in sending all incoming messages to all outgoing connections save by the one the message came by, and storing an identifier for the request to avoid duplicates. It is conceptually simple, but not very efficient, as every message can arrive to a given node several times.

In [1] an efficient approach is considered. It states that, in an (n,k)-arrangement graph a message can be broadcasted in at most O(k log n) steps. It also guarantees that the message is received by every node exactly once. But it relies on the properties of the arrangement graphs, a special kind of graphs whose topology is known in advance. We can not guarantee that. Even in the case that the FADA rearranged it structure to become an arrangement graph, the algorithm is based on reliable connections, as a given message is not sent by two separate ways to arrive at the same destination. But we've seen our connections are not reliable, and the behavior of the FADA should not rely on that circumstance.

[2] approaches the same problem in a different way. It considers the graph to be a star or pancake network, a special case of Cayley graphs. These graphs are hierarchical, which forces a structure, and this is not applicable to our case.

[3] did not force any structure, but relied on an already existing broadcasting medium. It only serves as an optimization of the operating system broadcasting primitives.

In [4] no structure is forced in advance, and the network can be dynamically configurable, and needn't be reliable. But unfortunately it asks nodes to know the immediate neighborhood, assuming that getting that information is easily obtained at a low cost, and fast. In our case, searching that information (neighbors up to two hops away) is much slower, and gets in the way of doing an efficient job. To perform an efficient broadcast an inefficient information gathering must be achieved. No gain is obtained.

Finally, in [5] we find the same objections we already considered in [1] and [2].

Put in other words, to perform efficient message broadcasting it is needed to avoid duplicates of these messages arriving at the same destination. To ensure that, it is needed to know which messages will arrive at what destination. In short, it is needed to know (at least part of) the structure. But the high availability design of an unhierarchical and unmaintained structure avoids any prior knowledge.

FADA behavior

This section explains the mechanisms used by the FADA architecture. These mechanisms deal with two goals: the capability to store and retrieve service proxies, and the ability to manage the structure of the graph of FADA nodes.

Once a FADA node is set up it can be used to store service proxies. Service providers will offer a service proxy for their service/s. This proxy must be written in Java, but the service provider can use whatever implementation for its service (either Java RMI, HTTP, JNI and some other language, JXTA, etc.), always keeping in mind that those proxies will be downloaded by clients and will execute from the clients' Java Virtual Machines (JVM from now on). This means that clients might be computationally poor machines, and the service proxy is expected to work even under those conditions. Also, the network connection of the client might be of low performance. This must also be taken into account. The service proxies should be simple and small.

If the service provider uses the FADA Toolkit class FADAHelper it is very easy to register a service proxy. The lease housekeeping is done by this class, so the service provider need not to fiddle with it.

In a similar way, a client doesn't have to know how FADA is implemented and what are its methods. Instead, it can use the FetishImpl class methods to look for a service implementation.

The flooding algorithm consists in the following:

- A node starts a broadcast: it sends the broadcast message through all the known connections.
- A node receives a broadcast message by any connection: it keeps the message identifier, then sends the message to all known connections except the one the message came from.
- A node receives a broadcast message whose identifier is already in the list of identifiers: the message is dropped silently, because receiving an already received message means there is a cycle in the structure. By dropping the message we make sure we avoid infinite loops.

Service searches will be triggered by a client in a known node. This node will do local lookup, and will also extend the search to all known neighbors, using the flooding algorithm. To allow fast response, the

lookup results from every node are NOT sent back in the same way that lookup requests came by (called routing). Instead they are directly reported to the interested node. Also, the local node lookup and the lookup extension is done in parallel by the use of threads.

To avoid overloading the FADA node that started the lookup with responses, there are some ways to limit the number of responses sent to a node. The first one is the distance limit. A lookup request will only be sent to nodes that are as much a predefined number of hops away from the origin of the lookup request. This limits the radius of the area where the lookup request is sent.

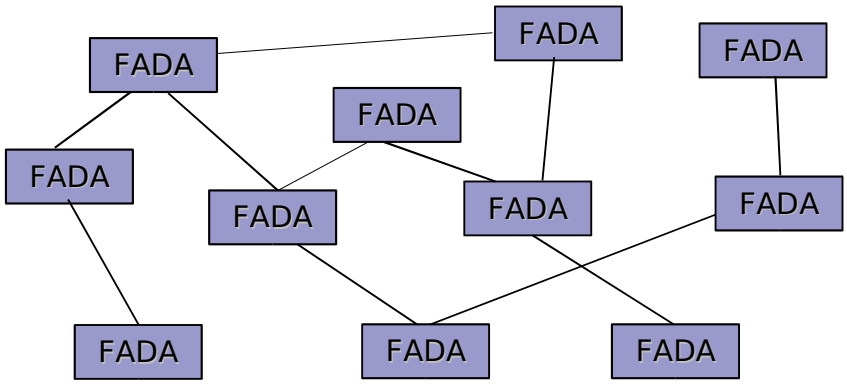
The second one is time. Once a maximum number of milliseconds have passed since the starting of the results recovery the results structure will be returned and destroyed. Any additional response from a remote node will be discarded. This doesn't avoid incoming connections, but avoids memory flooding in the origin node. A virtual global time is used by every FADA node. FADA nodes use ntp servers to synchronize their virtual clocks.

The third one is space. Once a maximum number of responses have been received the results structure will be returned and destroyed. Any additional response from a remote node will be discarded. This doesn't stop the broadcast mechanism nor avoids incoming connections, but avoids memory flooding in the origin node.

Service Directory Architecture

Service proxies are found and resolved by the FADA. Taking into account that only one FADA node is not enough to maintain all the information of availability of multiples services, a graph topology architecture of FADA nodes is proposed to search, store and recover services in an efficient way.

shows the FADA architecture. The structure is a graph model. This structure can be composed of any number of FADA nodes located on different hosts.

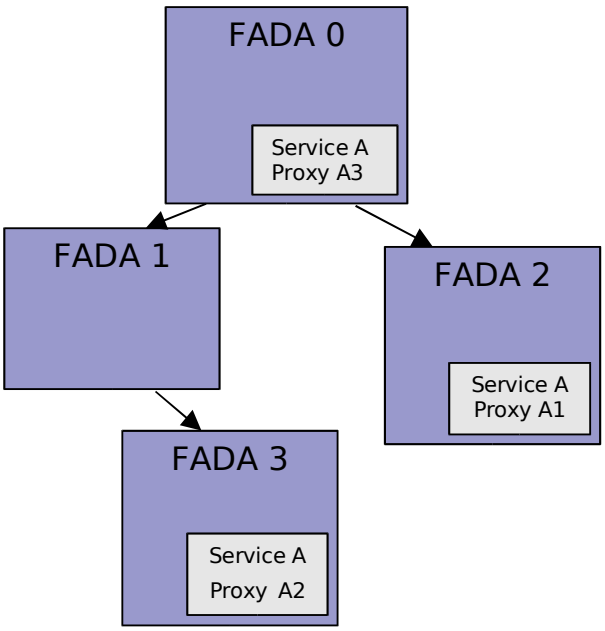


FADA Architecture

Any service can be registered in any FADA node of the architecture and any client can search for a service starting the search in any FADA node.

Next figure is a simple example, in order to explain how this structure works. *A* represents *HotelSearch* Interface and *Ai* represents the proxies of different *HotelSearch* Service implementations. For example, *A1* would represent the proxy of one specific service implementation of the *HotelSearch* Interface. When a service provider wants to register a new service, it interacts with any FADA.

Example: A service provider wants to register a new proxy (*A2*) for the *HotelSearch* Interface, and discovers and joins the federation though the FADA node 3. The FADA node 3 stores internally the *A2* proxy.



Architecture Example

When a client wants to search for an implementation of one interface, interacts with any FADA. The FADA will carry out all the procedures in order to return to the client the requested service. First of all, you must notice that the FADA may return more than one implementation (proxies) to the client. The provided result is a FadaServiceMatches object, that includes pointers to the service proxies. The FADA will check if it contains the requested proxy. If it contains a proxy it will be part of the result.

After that, the FADA will expand the search process to all direct neighbors, which in turn will check if they contain the requested proxy, and will also expand the search process to all neighbors but the one the search request came from (nor the original FADA). This prevents direct search cycles.

Every FADA that receives a search request take note of the search identifier, FadaSearchID, and will reject all search requests with the same FadaSearchID. This prevents transitive or indirect search cycles.

Example: A service provider wants to search implementations of a service (*A*) for the *HotelSearch* Interface, starting from FADA 1. The

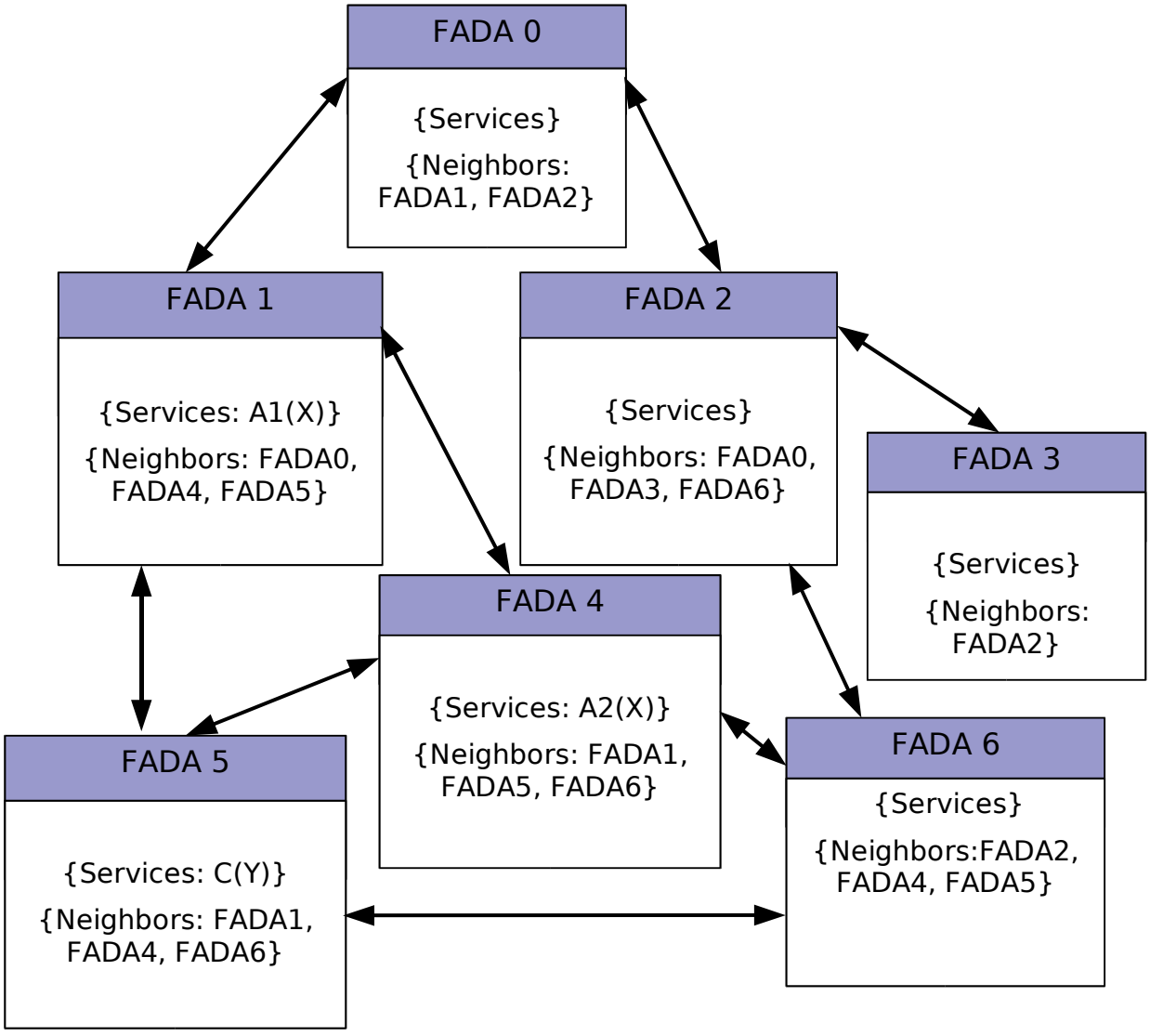
FADA 1 checks if it contains some proxy. The FADA 1 does not contain any proxy to match with the requested service. FADA 1 propagates the search through its neighbor FADA 1.1. FADA 1.1 repeat the same process. At the end, A1, A2 and A3 are found by the system and is returned to the Client. All search matches are returned directly from the node that found the reference to the node that started the lookup process. This allows a fast response.

The search process will end when the requested number of responses is reached (neighbors have no way to know this, but the node that started the search will reject any responses past this number) or the specified number of milliseconds from the start of the search is reached.

The number of the FADA nodes can be increased according to the needs of the registered services. This architecture maintains the main properties of the Jini architecture. Having more than one FADA node avoids an overload of services in any node.

Graphical overview

The next graphic shows the FADA architecture as well as the information that must be stored in each FADA to maintain a graph structure.



Detailed Architecture Example

Diagram description

- **FADA 1** : Identifier for the FADA node.

- **{Services: A1(X)}** : Indicate that there is the A1 service (that implements interface X) registered in the FADA node where appears this indication.

- **{Neighbors: FADA0, FADA4, FADA5}**: Indicate that this node knows the location and can therefore contact and cooperate with the FADA nodes identified by the identifiers FADA0, FADA4 and FADA5.

Searching service example

When a FETISH client wants to search for an implementation of one interface, interacts with any FADA. The FADA will carry out all the procedures in order to return to the client the requested service proxy. Firstly, you must notice that the FADA may return more than one implementation (proxies) to the client. The provided result is a set of proxies. The FADA will propagate the search towards the neighbor FADA nodes.

Example: A FETISH provider wants to search implementations of an X Interface, starting from FADA1. He will call the method lookup in the FADA1. This method will create a new unique SearchID (96 bits long), start the lookup within itself and, in parallel, will extend the search to neighbors FADA0, FADA4 and FADA5. FADA1 will find a service A1 that implements interface X, and will add the result to the result set, identifying the result with the search ID. At the same time, FADA0 will do the internal lookup and the extension to FADA2 in parallel. It won't find any reference to a proxy implementing X, so it will give up, telling FADA1 about 0 results found. Meanwhile, FADA4 has started the local lookup and extended the search to nodes FADA6 and FADA5. The latter will be rejected by FADA5, as it has already received a lookup request with the same SearchID by FADA1 (transitive cycles avoidance). FADA6 will do the parallelized lookup within itself and extension in FADA2, which will also reject the lookup request, as it will have received a request with the same SearchID via FADA0. At the end all nodes have been searched, and all responses (A1 from FADA1 itself, A2 from FADA4) are sent directly to FADA1, which, after reaching the requested number of responses or having

timed out, will return the results to the FETISH provider that started the search process.

Had the FETISH provider specified an nHops parameter of 1, no search would have been extended beyond FADA5, FADA4 and FADA0, as the parameter nHops is decreased in every lookup extension. So service A2 would have never been found. But that might be right, as the potential number of responses could bring down the FADA1 if there were no time, distance and/or space constraints.

FADA communications layer

FADA has evolved from being an extension of the Jini reference implementation, trying to overcome its inherent limitations, to its actual state, a complete rewrite of the software from scratch dedicated to deal directly with the roles it has to accomplish. Such evolution can be observed from the various methods used for communication among FADA nodes and between the FADA and FADA clients.

Communication among FADA nodes and between the FADA and FADA clients started by using the standard Java Remote Method Invocation framework (Java RMI). This method offers several advantages:

- The use of an already established standard, with its complete set of tools and classes.
- The easy integration with the mechanisms used by the Jini reference implementation.

Unfortunately, some drawbacks promptly appeared. The most notable of them are:

- The use of a wire protocol that can not be used as is in environments that involve the crossing of a firewall.
- The need to install an additional wrapper to cope with such environments.
- The lack of control on the wire protocol itself. This lack of control precludes the use of different communication mechanisms to ensure privacy, for example.

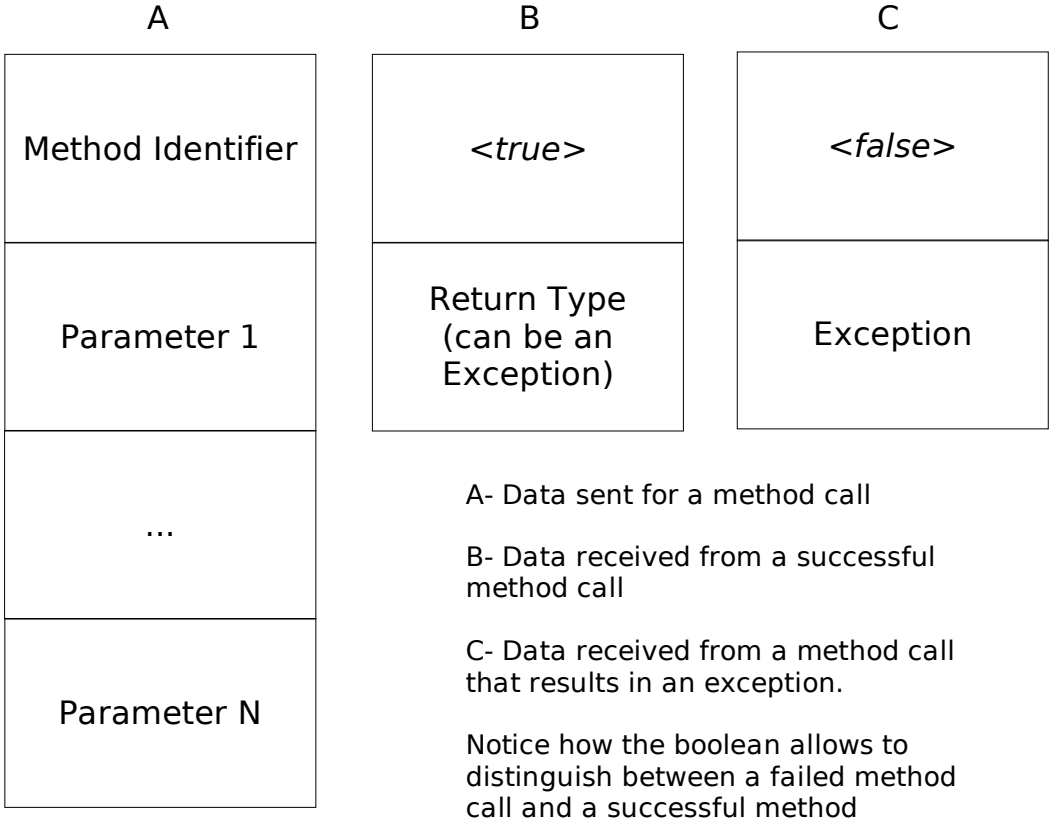
The first shift from the RMI environment was the use of XML-RPC calls through HTTP. XML-RPC settled the foundations for the now widely used SOAP mechanism. SOAP stands for Simple Object Access Protocol. It is heavily based on XML.

The goal of the FADA protocol is to be an alternative to the default RMI protocol, not to be the most interoperable protocol. Interoperability is not a concern within the FADA, because the FADA nodes have no need to be interoperable with other services. The FADA proxy itself (and service proxies in general) provides the needed interoperability layer, so there's no need to lower the

interoperability level down to the wire protocol. Moreover, SOAP incurs in an overhead two to three orders of magnitude above the straight RMI protocol.

Therefore it was chosen not to adopt SOAP, but rather its earlier version XML-RPC. This Remote Procedure Call protocol doesn't need DTD's or XML Schemes to validate, and it's quite simple and straightforward to implement. Its use of ASCII XML documents make it easy to transfer data in HTTP messages, and the use of HTTP is a Good Thing from the end user perspective, who will have to deal with firewalled environments that provide an HTTP proxy to access the Internet.

XML-RPC, though, is not readily usable for the FADA purposes, so all the classes involved in FADA communications had to be represented with the XML-RPC data primitives and constructors. This lead to the appearance of a whole package of Java classes to transform Java and FADA classes to/from XML-RPC data types. Furthermore, an XML parser had to be chosen. It was decided to adopt the TinyXML parser, which is small and yet powerful enough to provide the required functionality.



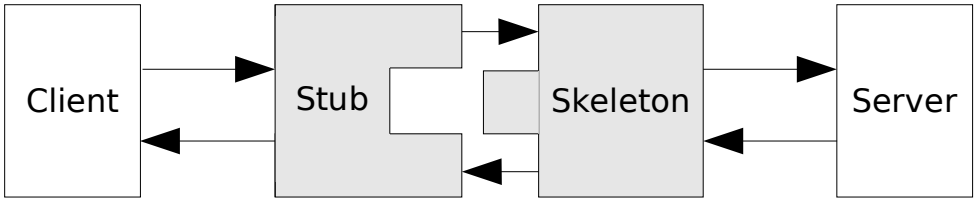
After a period of successful operation it was decided to give yet another turn to the screw, and drop usage of XML at all. Although small, the use of XML RPC still imposes an overhead on all FADA communications, and the use of the wrapper classes to transform between FADA types to XML-RPC types makes it difficult to add new classes needed in communication. A binary mechanism was the ideal form, and that is the last step in the development of the FADA communication layer.

RMI and the new FADA communications layer

RMI is the Java Remote Method Invocation. It is a framework that allows Java classes to call methods on Java classes that do not reside in the same memory space. This is suitable for intercommunication between classes that reside on different Java Virtual Machines in the same host or in geographically separated Java Virtual Machines.

RMI uses a pair of classes, named Skeleton and Stub, to perform communication between a server and a client in a more or less transparent way. In Java 1.2 the skeleton part was dropped, and its functionality embedded in standard Java Remote Objects. The skeleton/stub pair is an inheritance of the homonymous entities used in RPCs, the Remote Procedure Call framework also invented by Sun.

RMI remote objects implement an interface that extends the standard Java interface Remote, from the package java.rmi. Classes that wish to be used as remote objects from within Java, using RMI, have to implement an interface that extends java.rmi.Remote. Such objects can be passed onto a command line tool called rmic, the rmi compiler. This rmi compiler generates the pair of stub/skeleton classes. For each different method call provided by the remote interface (the interface that extends java.rmi.Remote) an id is generated.



1- The client calls the interface method on the remote object (really the stub)
2- The stub serializes parameters and creates the data stream,
that is sent to the skeleton by the means of the transport layer.
3- The skeleton reads the data stream and, depending on the method id,
deserializes the parameters.
4- Once deserialized, the skeleton calls the server method with the proper
parameters. Method overloading is provided by the means of different method ids.
5- The server performs the method execution.
Return types or exceptions are returned back to the skeleton.
6- The skeleton returns the results back to the stub.
7- The stub deserializes the data stream and returns the results to the client.
If communications errors arise the stub wraps them in a generic
RemoteException that is catched (or thrown) by the client method.

The stub implements all remote methods in the remote interface. This stub serializes the parameters for the call and sends them, together with the method identifier, to the network endpoint where the skeleton is listening.

The skeleton, in turn, sits waiting for requests. Whenever a new request is received the skeleton deserializes from the stream the identifier for the method, and then deserializes the method invocation parameters, according to the method called. When all data has been obtained, the homonymous method in the server object (which is now local to the skeleton Java Virtual Machine) is called. Upon termination of the method call, the skeleton receives the return type (or Exception, if such is the result) and serializes them back to the stream where the stub is waiting for the return. The stub then deserializes the return type (or the Exception) and gives it back to the caller. The caller has performed a remote call almost transparently.

However, the communication details of this approach are hidden from the programmer's perspective, which eases the implementation of classes that use RMI, but leaves little room for optimization or other kinds of manipulation. Therefore, if the standard underlying mechanisms are not suitable for a particular application, there's not much that RMI can do for the programmer. The programmer must invent its own method of remote invocation that is suitable for her scenario.

The same mechanisms used in Java RMI have been applied to the latest FADA communications layer, though through a different set of classes. The greatest benefit of the FADA approach is that the transport layer, that moves the data between the skeleton and the stub is now exposed. That means that the developer can plug its own transport mechanism. This user-definable transport mechanism can use whatever means for compression, encryption or otherwise obfuscation of the transmitted data, at the transport level. What's more, the server part (the skeleton) can now multiplex several communications in the same socket, because the decision on what skeleton to call can be made by the transport layer. Or maybe the skeleton and stub pairs may choose not to use sockets at all, if the developer likes it so. No transport means are enforced, although a default HTTP/binary-stream data flow can be used in lack of any other. By using HTTP the caller may be behind a firewall and transparently use an available HTTP proxy, without any additional modification of the software.

The identifiers for the interface and method ids have been generated by using the Tiger hash function, proposed by Ross Anderson and Eli Biham in 1996. More information on the Tiger hash function can be found in

`http://www.cs.technion.ac.il/~biham/Reports/Tiger/tiger/tiger.html`

FADA mechanisms

Here we will depict the basic FADA mechanisms to make a cloud of FADA nodes become a single virtual Lookup Server.

Discovery mechanism

Discovery is the only part of FADA that FADA doesn't do itself. After all, it's pretty difficult to obtain functionality from something if an initial interaction is not performed in advanced.

FADA doesn't force the user to use a discovery mechanism. If the user knows the URL of a FADA node it can be used to join the federation in that point, and by using the FADA methods regarding the directory structure it is possible to obtain the location of the rest of FADA nodes. However, FADA provides with a way to perform discovery of FADA nodes if no URL is known in advance. The mechanism is based on the Domain Name Service, the DNS.

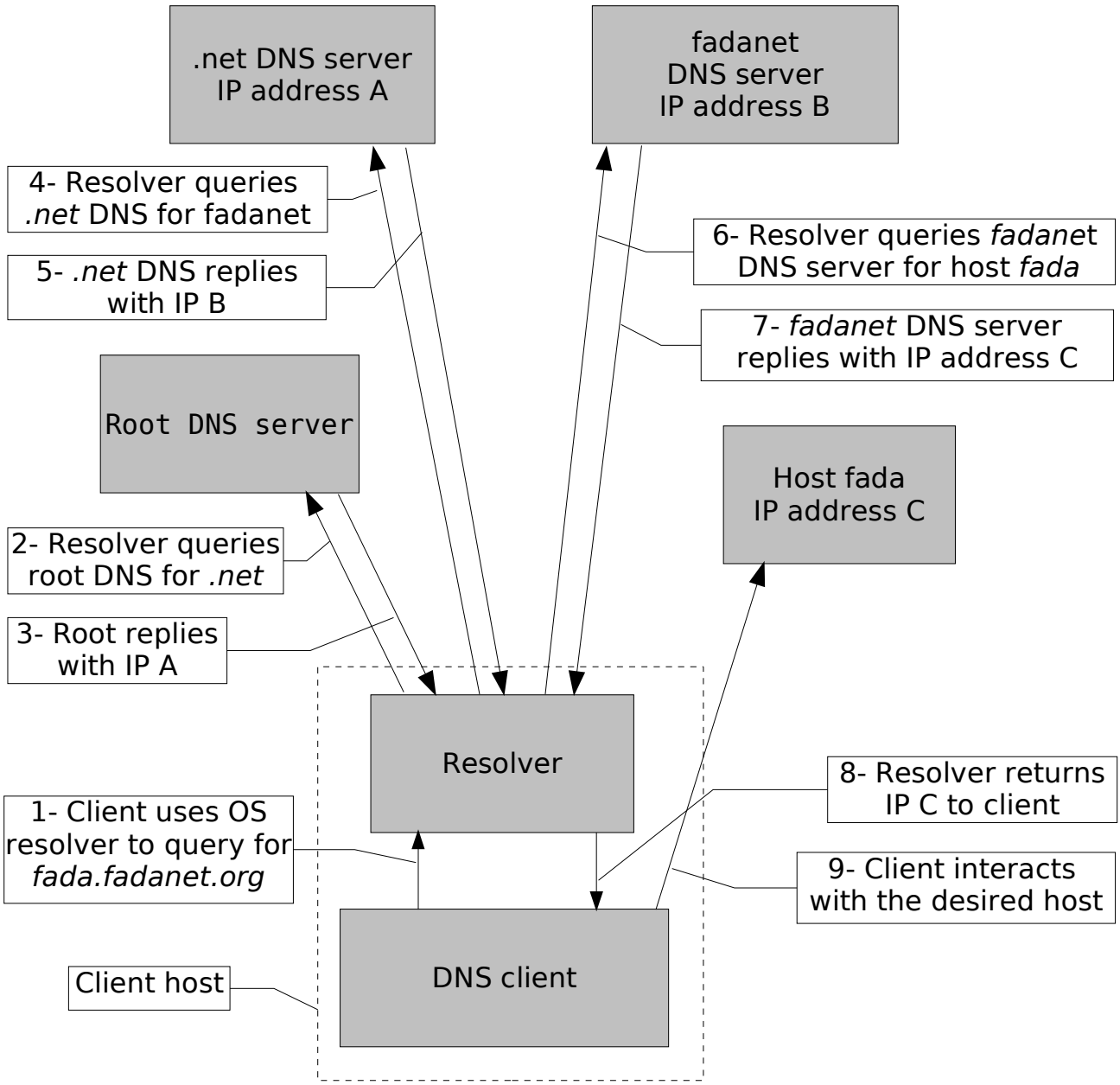
A DNS server performs the mapping between domain names and IP addresses. The DNS service follows a hierarchical structure where there is (at least) a root DNS server, and that DNS server knows the IP addresses of the DNS primary domain servers. A DNS primary domain is the trailing part of the dot separated domain names. For example, in the URL fada.fetishproject.com, the primary domain is .com.

Each one of the primary domain servers know the IP addresses of several secondary domain servers. In the example above, the secondary domain is fetishproject. Each one of the secondary domain servers know the IP addresses of tertiary domain servers (if such tertiary domain exists), and so on until the domain part of the URL is exhausted and all that is left if the host part, which is a single host name (fada in the example above).

The DNS server for the last domain obtained (fetishproject in the example) knows the IP address of the host itself. When a user polls the DNS service for a URL such as fada.fetishproject.com the DNS servers involved are: the root DNS server, the .com domain DNS server and the fetishproject domain DNS server. The DNS server for the domain fetishproject returns the IP address of the host fada.

This, in itself, is short of amazing. What is interesting is that every DNS server mentioned in the above paragraph can be replicated. That is, there may be two DNS servers for the domain fetishproject, for example. This replication is transparent for the end user, and allows the DNS to be a high availability service.

What's more, a DNS server can be configured to give different responses in successive calls. That is, the DNS server for the domain fetishproject can respond a query for the host "fada" with different IP addresses every time it is polled. In this way, several hosts can be given different IP addresses, and all of them can be registered in the fetishproject DNS server (or servers, as this server can be replicated) under the same name.



This allows the FADA to have several FADA nodes whose IP addresses are registered in the DNS server for the domain fetishproject. If a user requests the IP address of the host fada.fetishproject.com the DNS service can respond with an IP address to one request, and with a different IP address the next time, and so on, until the total number of hosts have been visited, and the IP addresses obtained start repeating.

If the lease mechanism is applied to the DNS service it is possible for FADA nodes to register in the DNS service and to keep a lease on it to maintain the DNS mapping from the name fada.fetishproject.com to their IP addresses. The moment a FADA node crashes the lease will expire, and the DNS service will silently drop it from its mapping table.

This, together with the replication capabilities of the DNS service makes the FADA a highly available, management free service.

Registration mechanism

The registration mechanism is the way by which a service proxy is sent from the service provider to a FADA node, and this FADA node keeps a copy of it, with the information needed to find it later, and to successfully return it to the requester. The service provider acts as the client of the FADA architecture, and the FADA acts as the server side from the service provider's point of view.

Client (Service Provider) side

The client side must construct an instance of the class FadaServiceItem. This class contains the service proxy to register, the array of entries and the desired FadaServiceID. If no FadaServiceID is desired this field may be left empty by specifying null. If no entries are desired to be specified this parameter may also be replaced with null. Upon construction the class FadaServiceItem will introspect the service proxy and gather the complete list of implemented interfaces, either directly or through interface implementation and/or inheritance from class extension. This list of interfaces (as an array of java.lang.String) can be checked through the available methods.

An entry point to the FADA must be obtained. Creating a FadaProxy with no arguments defaults to the url fada.fetishproject.com:2002. It is recommended to obtain the list of neighbors of this node and perform this operation recursively a random number of times to obtain the url of a random FADA node in the architecture, so as to avoid overloading the FADA node at fada.fetishproject.com:2002 with service proxies. In the future the number and size of registered

proxies in a FADA node may be limited, but only after study of the service proxy population. Before this happens the FADA must be widely used.

The caller issues the method invocation request on the FADA node associated to the FADA proxy. The request will be processed by the FADA node (see below) and a response to the HTTP POST request will be obtained, containing an response with the resulting FadaServiceRegistration, which contains the FadaServiceID obtained (that can be the one specified, if accepted, or a completely new one, if none specified).

The FadaServiceRegistration also contains the FadaLeaseProxy (that has been constructed locally, it hasn't been serialized from the server side) that will allow the client side to renew the lease and so ensure the service proxy is not deleted from the FADA after some time.

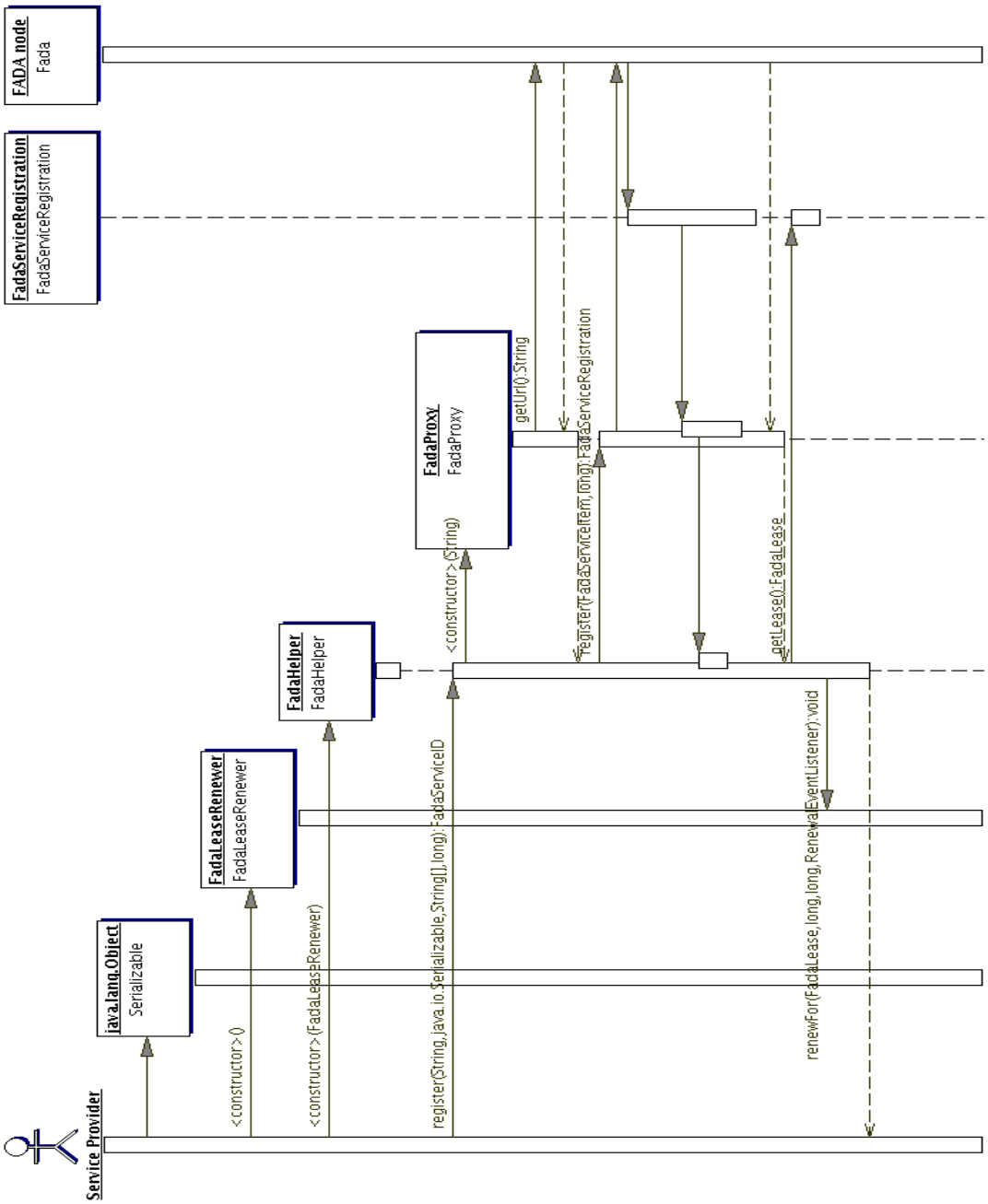
Server (FADA) side

When the FADA receives an HTTP request on the port it's listening to it spawns a new instance of the local HttpProcessor implementation. This HttpProcessor reads the entire HTTP message (using the classes provided in the package net.fada.http, such as HttpHeader and HttpMessage), and checks if it's a POST request, a GET request or any other kind or request. In the latter case an error is issued in return (unsupported request). If it's a GET it refers to the web interface to the FADA node, and it's executed by the method processGET(). If it's a POST it's executed by the method processPOST(). In the case of registration the method does the following:

– It creates a FadaServiceMatch instance, to speed up the lookup procedure, and also to separate the service proxy from the FadaServiceMatch. In this way the two-step lookup mechanism is easier to perform. The list of interfaces is dumped onto a TreeMap, so lookup complexity will only be logarithmic when matching the interfaces. The same is done for the list of entries.

– It computes the expiration time by adding the actual time (System.currentTimeMillis()) and the accepted lease time (which is the specified lease time chopped to a maximum value specified in the FadaLease interface). Then it adds the FadaServiceItem to a MinHeap. The lease expiration mechanism then takes place.

The UML sequence diagram that describes the registration mechanism is the following:



Lease expiration mechanism

Client side

The client side is not related at all with the lease expiration mechanism. This process is completely inner to the FADA node.

Server

The FADA node has an instance of Thread dedicated to the task of watching the expiration times of the registered service proxies, and the deletion of these proxies when their leases expire.

This Thread will fetch the expiration time of the root of the heap, which is the least value of the whole heap (this condition is assured by the properties of the min-heap), and check how much time is available until the lease expires. When this time has been computed it calls the wait() method for the amount of time calculated on an Object instance that serves as a lock for all concurrent threads within this task. When this this time has passed and the Thread instance awakes again it deletes the root of the heap, fetches the expiration time of the new root and repeats the process. The deletion of the root takes a time proportional to the logarithm in base 2 of the total number of elements in the heap, and therefore is very efficient. Heaps are often used to implement priority queues, and the list of expirable items in the FADA node is a priority queue.

Lease renewal mechanism

The lease renewal mechanism responsibility lies on the client side, but it affects directly the server side, so both parties are described here.

Client side

The client side (service provider) of the lease renewal mechanism is responsible for renewing the lease before it expires. Network delays must be kept in mind when performing this task, because the lease renewal request may be late to the destination even if it was issued before the lease expired.

The FadaLease offers methods to known the expiration time (which is calculated as the time the registration response or the lease renewal request response was received at the client side plus the granted lease time). By taking the actual time before and after the registration or lease renewal attempt and substracting them an estimation of the real network delay value can be computed. This

value is very unstable, so a filtering mechanism must be performed to gain reliability.

The FadaLeaseProxy may contain one or more instances of an implementation of the discrete linear Kalman filter to obtain a more reliable value for the network delay. This set of discrete linear Kalman Filters is embedded in a class called LeaseDelayFilter. Thanks to the properties of the Kalman Filter it is possible to make a very reliable (though not fail-proof) estimation of the next network delay value. This is possible because, although network delay may dramatically vary in a short period of time, usually the network has a sustained average value (subject to changes, of course). This estimation takes into account the calculate variance of the measured network delay.

In practice it is enough to substract this estimation of the delay to the expiration time of the lease to know at what time the lease renewal request must be issued. The class FadaLeaseRenewer does exactly this, only that it keeps a pool of LeaseDelayFilter instances (stored in an instance of FilterBattery), and each of them can be shared among several FadaLease instances if they belong to the same FADA node (because the network delay will follow the same distribution). The class FadaHelper needs an instance of FadaLeaseRenewer to be instantiated, and so this mechanism is also applied when the registration and lease renewal issues are left to the FADA helper classes.

The estimation of the network delay is performed through the use of a set of mathematical tools known by the name of Kalman filter. In 1960, Rudolf E. Kalman published a paper describing a recursive solution to the discrete-data linear filtering problem.

The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) solution of the least-squares method. The filter is very powerful in several aspects: it supports estimations of past, present, and even future states, and it can do so even when the precise nature of the modeled system is unknown.

This approach is what we needed: a mathematical model that was able to predict the future state of the network. Not only the Kalman filter gives the best estimation for the next value, but it also gives the error variance, which is much more useful in our case, as we do not need the exact predicted value, but an upper bound for that prediction. In other words: we don't need to know what the most probable next delay time will be, but what is the confidence interval for that value, and use the worst value as our estimation.

The Kalman filter as-is, though, did not fit our case, as it makes some assumptions that didn't hold true. It assumes the real value is disturbed by a white noise signal. In our case the delay evolution of the network is not a continuous function, but we can model it as an average continous function disturbed by another function. This other function doesn't follow a gaussian distribution. Further investigation led us to a different approach to the use of the Kalman filter.

The model used in the FADA is the following:

- The delay is assumed to be a unknown constant. All known network delay models take in account the network topology, the enqueueing schemmes, the timeout counters, etc., but in the Internet it is not possible to have a global knowledge. Therefore, for lack of a better model, the estimation of an average unknown value will be the best possible effort.

- The instant delay (that is, the delay measured in a concrete instant of time) is assumed to be distributed around this mean value, following a Gaussian distribution.

The equations used are the general Kalman equations particularized for a one-variable case with a constant evolution over time of the parameter to be estimated, that is, the average delay in the first case, the average positive increment in the second case. The equations for both filters are the following:

- Initial values:

```
public DiscreteKalmanFilter(
          double A,
          double B,
          double Q,
          double R,
          double H,
          double initialX
) {
          this.A = A;
          this.B = B;
          this.Q = Q;
          this.R = R;
          this.H = H;
          this.xHatAPriori = initialX;
          this.xHat = 0;
          this.pAPriori = 0;
          this.K = 0;
          this.P = this.Q;
}
```

The parameter A is the relation between two samples of the estimated variable. As it is estimated to be a constant, A is 1.

The parameter B is the weight of the control input. As in our model there is no control input, it is assumed to be zero.
The parameter Q is the standard deviation of the process noise. It has been estimated (through experiments) to be 0.00001.
The parameter R is the standard deviation of the measurement noise. It has been estimated to be 0.01. In our model all the variation is assumed to come from the measurement.
The parameter H is the relation between the state and the measurement. As in our model the state IS the measurement, H is 1.
The parameter P is the error covariance. It is started with a value equal to Q.

```
public synchronized void start( double initialX ) {
          this.xHatAPriori = initialX;
}
```

The second method allows us to reset the filter without the need to create a new instance.

- Update (projection) equations:

```
public synchronized void update( double u ) {
          this.xHatAPriori = this.A * this.xHat + this.B * u;
          this.pAPriori = this.P + this.Q;
}
```

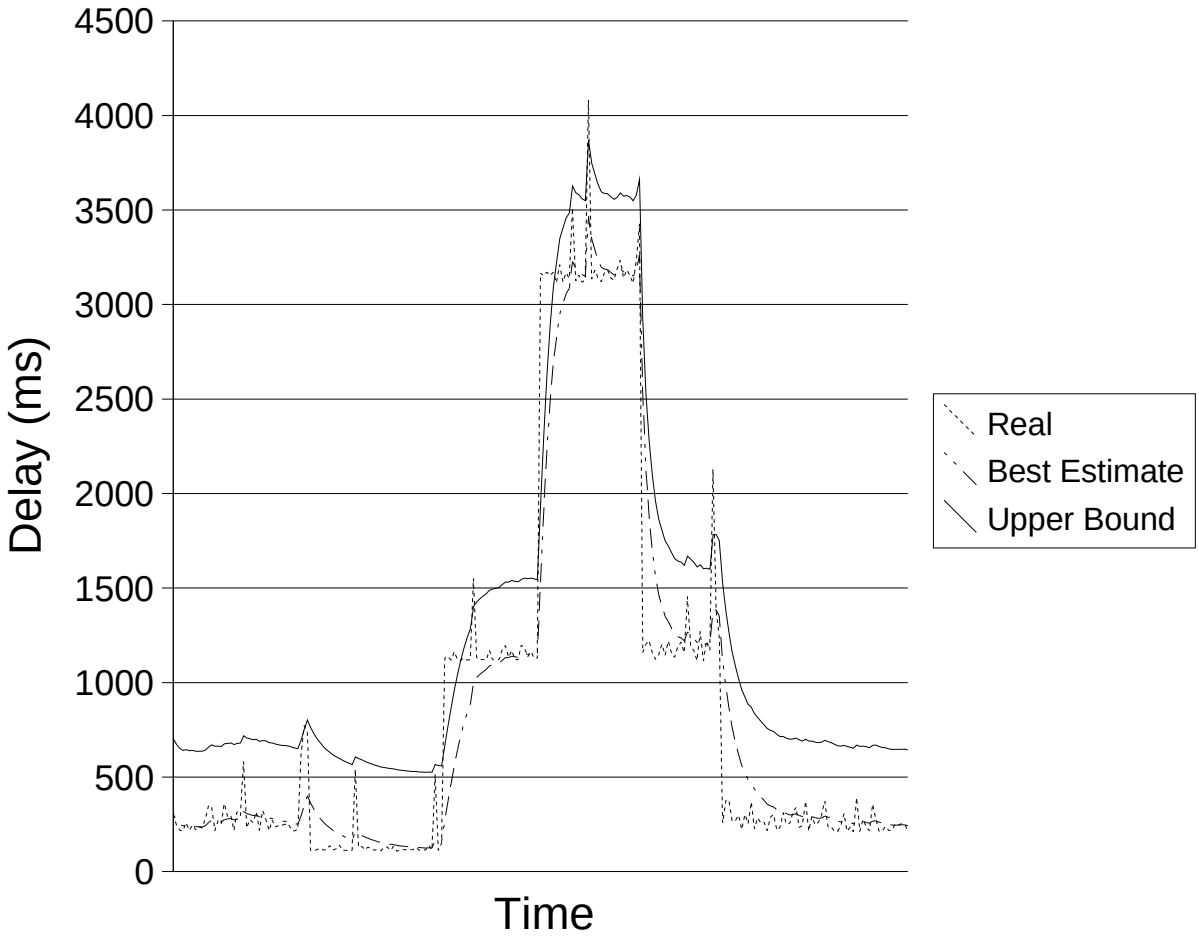
- Measurement (correction) equations:

```
public synchronized void measurement( double z ) {
          this.K = this.pAPriori / ( this.pAPriori + this.R );
          this.xHat = this.xHatAPriori + this.K *
                    ( z - this.H * this.xHatAPriori );
          this.P = ( 1 - this.K * this.H ) * this.pAPriori;
}
```

- Estimation

Instead of having one filter, we now use two. The first one estimates the average (mean) value of the network delay. Although the network delay is not a constant, its average value can be assumed to be constant, and the fluctuations around this central value may be assumed to be due to the variance. To get a more accurate model, this variance around the central average value must also be estimated. But we are not interested in getting an accurate model of the network delay. What we really need is a pessimistic estimation that tells us how worse can things go, and therefore we use another Kalman filter to estimate the average positive increase in network delay. Although not an exact model of the distribution or worst value

of the network delay, this approach did much better fit it. Simulation and experiments have shown a much better response than simple averaging: the failure rate of the modified Kalman filter was much lower. We call a failure a lease that couldn't be renewed in time.



The figure shows the behavior of the filter we're using when facing Internet delays. We implemented a simple server that received a request of a client, waited a random time (whose distribution was uniform around ±30% of a constant time, that was modifiable at run-time) and then issued the response. The spikes correspond to the Internet delays. The filter adapts quickly to new conditions on the network, although not so quickly that it would follow the previously sampled value, which would render the filter useless.

Even with the Kalman filter we still needed to ensure that service proxy registrations where not deleted from the registry when the service was still alive. A **reregistration** method was added to the class FADA helper classes, that could be used by the class that automatically keeps alive the FADA leases. Reregistration method registers the service proxy again, but it keeps the same parameters: attributes (given by the service provider), the interfaces implemented

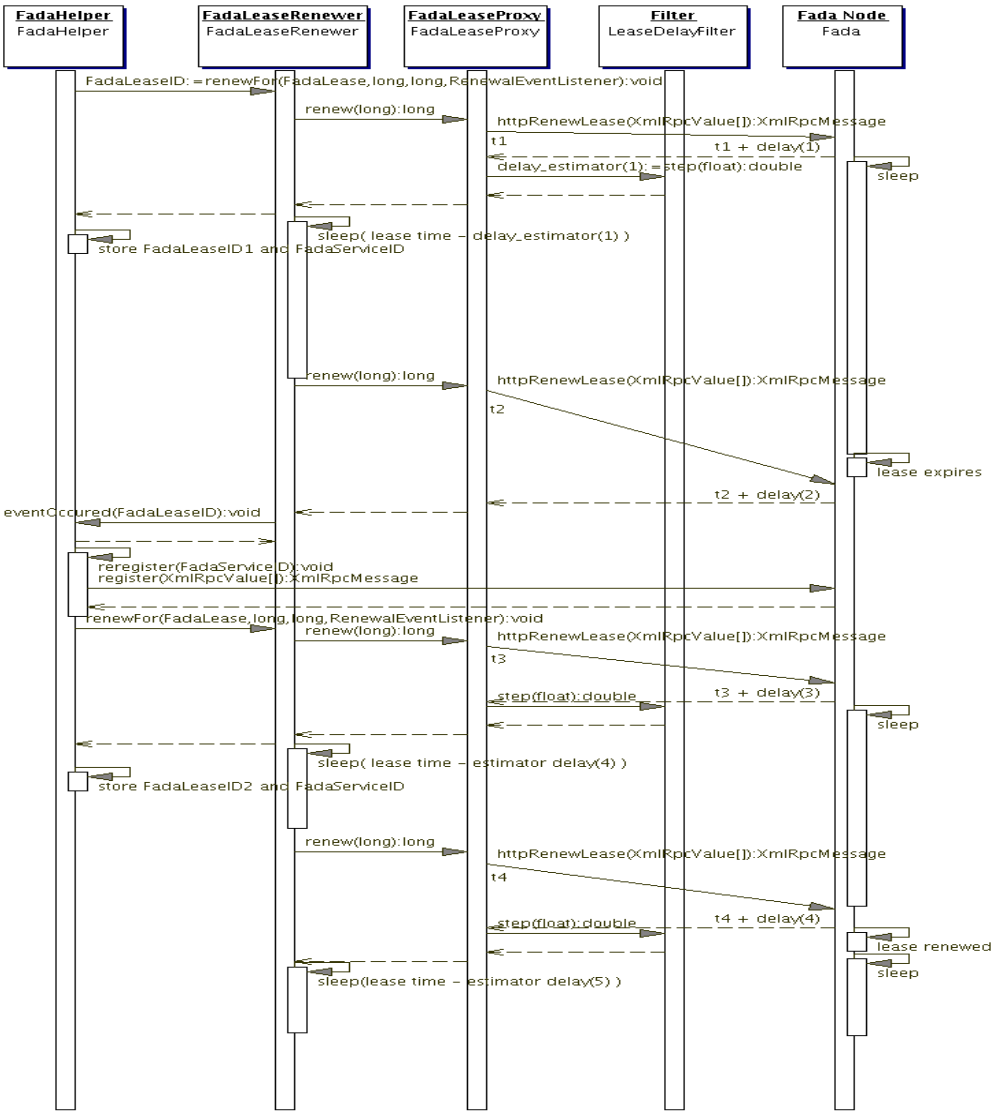
(given by the proxy itself) and ServiceID (given by Jini and FADA, but kept the same by FetishImpl and FetishLeaseRenewalManager). Even when a lease is not renewed in time, the state of the FADA remains stable, and users will only notice a delay in the response or a miss in the lookup, that will be fixed on the next lookup.

The FADA helper classes detect a FADA node failure by its lack of response in front a a lease renewal request. In that moment, the FADA helper classes residing in the server part of the revise locate another FADA node (mainly from a list of candidate FADA nodes obtained during the last successful lease renewal request) and register the service in one of them. In this way the registration is persistent even in case of FADA nodes failure. As all FADA nodes are idempotent (ie. non-privileged in any way) none of the service parameters are affected in any way. The state of the FADA network is stable in the long run.

Server side

On the server side (the FADA node), when a lease renewal request is received the lease is searched in a TreeMap that is put side by side with the min-heap. This TreeMap is sorted by FadaLeaseID, and contains a reference to the heap node that holds the FadaLease. With that data it is easy to remove the selected item, recompute its expiration time and reinsert the item in the heap. The Thread instance that makes the leases expire is notified of the changes via a call to notify() on the object that serves as a lock for all concurrent threads within this task. The Thread instance then awakes from the call to wait(), and initiates a new iteration to search the next lease that will expire. It is likely that the structure of the heap may have changed, because that’s the purpose of the lease renewal mechanism.

The UML sequence diagram that describes the lease renewal mechanism is the following:



Lease cancellation mechanism

The lease cancellation mechanism is, obviously, initiated by the client side, and performed by the server side (the FADA node).

Client side

The client side, having the FadaLeaseProxy, makes a call to the method cancel(). This method sends a remote method call to the FADA node where the FadaLease is sending as a parameter the FadaLeaseID of the lease to cancel. The result of this operation, if no transmission error occurs, is always a boolean with the value true.

Server side

The FADA receives the HTTP POST request that contains, as usual the method call. The HttpProcessor.processPOST method is invoked, and the FadaServiceRegistrar is notified of the intention to cancel a lease. The lock is acquired to avoid the Thread instance that expires leases to touch the structure before the lease has been cancelled. Then the FadaLeaseID is searched in the TreeMap structure (logarithmic cost), and a reference to the heap node obtained. This heap node is eliminated from the structure, and its place filled with the last element of the heap, which is sorted up and down again to ensure the structure is still a min-heap. That's all. Further attempts to renew the expired/cancelled lease will result in an exception.

Lookup mechanism

The lookup mechanism involves one client side, the service proxy client, and many (potentially all) of the FADA nodes available in the FADA architecture.

Client side

The client side (the end-user of the service proxy, be it a human user or another program, such as a combined service proxy) constructs a FadaServiceTemplate object with the FadaServiceID, the set of entries and/or the set of interfaces the service proxy must implement, as specified in chapter 6. This template will be passed as a parameter in the call performed on the server side (the FADA node).

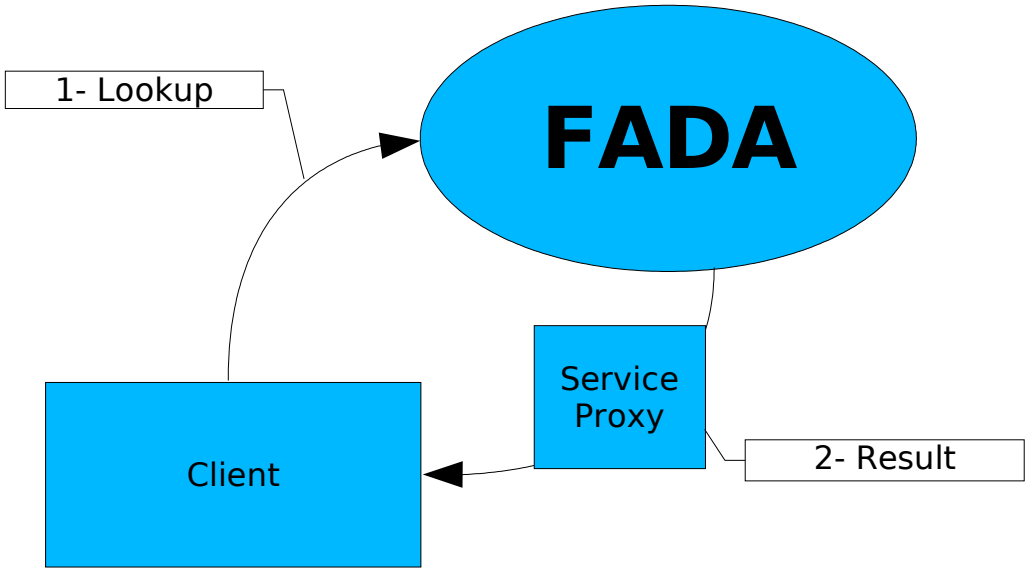
Server side

FADA nodes, when receiving lookup requests, will start a process to try and find matching results for the query on as many nodes as possible, sticking to the limitations specified by the received query. A

flooding algorithm is used to broadcast the query throughout (potentially) the whole FADA architecture.

To specify this process, FADA nodes can be divided in three categories, depending on the roles they are playing in this flooding procedure.

We will call initiator the FADA node that received the query from the client side, and will initiate the flooding algorithm to broadcast the query.



Every node that receives a lookup query and propagates it to other nodes we will call a propagator.

Last, we will call leaves those nodes that detect the conditions to stop the broadcast are met, and so stop propagating the query.

Those roles are not assigned at configuration time, but rather taken by each FADA node at runtime, depending on its connectivity state and the state of the lookup being taken.

Initiator

The initiator FADA node receives the lookup request from the client, generates a new globally unique identifier for this search, creates a structure to store the lookup results (identified by the freshly created globally unique identifier) and propagates the query to all known nodes, telling them also the identity (FadaID, that contains the url) of itself (the initiator) and also the FadaID of

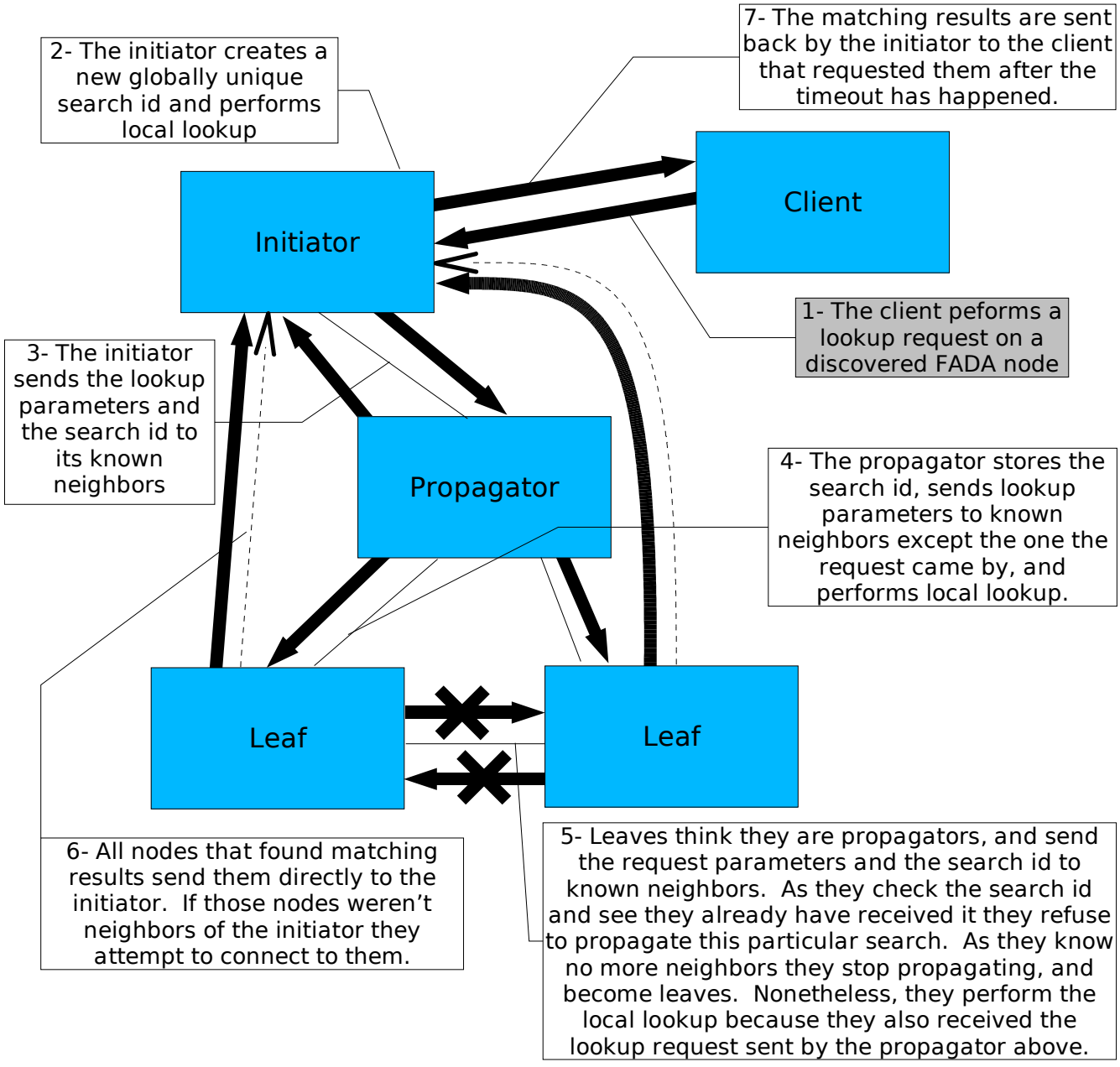
the node they must NOT propagate the query to (in this case, the initiator FadaID also).

Propagators

Propagators are FADA nodes that receive a lookup request from another FADA node. They receive the identifier of the lookup request created by the initiator, and they check if they have already received it. If it is so the received request is silently dropped. In this way transitive loops in the broadcast mechanism are avoided.

Propagators also receive the FadaID (which contains the url) of the initiator, to know where the potential lookup results must be sent to, and the FadaID of the FADA node they must NOT forward the lookup request. This last FadaID belongs to the FADA node that sent the lookup request to this propagator, and we will call the sender.

Then the propagator sends the lookup request to all neighbors except the initiator and the sender, because both of them already have received the lookup request, and have already forwarded it to other propagators. In this way direct loops in the broadcast mechanism are avoided.



Leaves

When a FADA node detects that the number of hops has been decreased down to zero, they don't propagate the query anymore. When a FADA node detects that the time the query should have been completed has passed, they also stop forwarding the query. FADA nodes can do the latter because they periodically poll the official time from a time server. They don't change the time of the

local host, but rather compute the difference between the local host time and the official time, and use that difference to get a hint of the official time. In this way they can know when a lookup request was meant to be dropped.

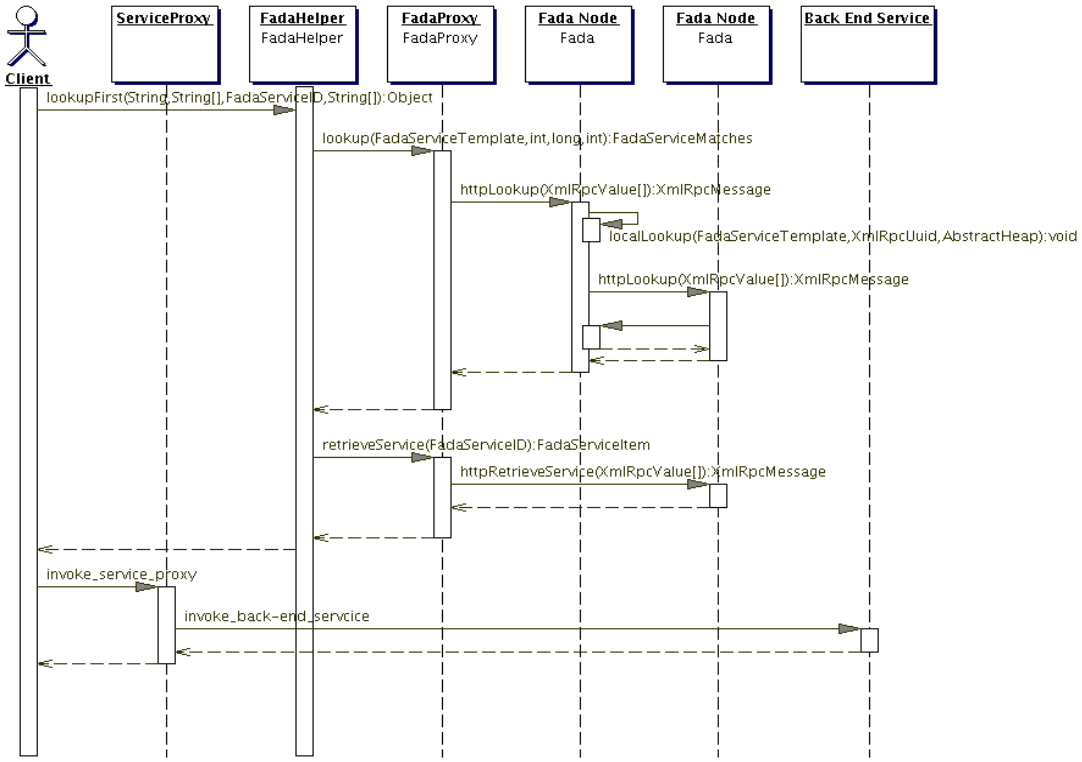
The third lookup extension constraint is the number of responses. The responsibility is left to the initiator, which, when the number of requested responses is reached, or any of the above two conditions are met, sends a response to the caller (the client side), with all the achieved lookup results. Propagators and leaves are, in the actual implementation of the FADA, unable to know if the requested number of responses has been reached or not.

Propagators keep the received search identifiers for several hours, to ensure that a query hasn't been traveling around the Internet and eventually arrive to a FADA node that has already received the query (there is no way a propagator can know if a neighbor of his has ever received the query).

No matter what role a FADA node involved in a lookup request is, they all perform (in parallel with the possible extension mechanism) a local lookup to match the received template against the registered service proxies. When they have completed it they send it to the initiator (the initiator uses local method calls, the propagators and leaves send it usual HTTP messages containing serialized data). The initiator uses the java.lang.Object monitor, the statement synchronized and the methods wait() and notify() to resolve the contention problem for concurrent access to the structure that will finally hold the obtained lookup results. Note that there will never be replicates in the results, because every node reports only local matches.

All FADA nodes that report lookup results to an initiator will also attempt the creation of a connection between the reporter and the initiator. What it means, they will become neighbors (if they weren't already). In this way high-connectivity is achieved, and the probability of failures in case of isolated nodes will decrease as long as these FADA nodes are used and have service proxies registered and lookup results to report.

The UML sequence diagram that describes the lookup mechanism is the following:



Note that it also depicts the retrieval of proxies, explained in the next section.

Retrieval of proxies

Lookup results contain only pointers to the real service proxies, and not the proxies themselves. In this way bandwidth is saved, and the response times are much shorter, because typically the vast majority of a set of lookup results will be discarded, and only one implementation will be used. This is not much of an issue for the Jini sample implementation, that is designed to work in a LAN environment, where the bandwidth is typically higher than when dealing with the Internet, and also the network delays are several orders of magnitude shorter. In the Internet, though, downloading two megabytes of data to use a service proxy and a class that both occupy fifteen kilobytes won't make the users happy.

The retrieval of proxies can be divided in two phases. The first one is discovery, and has been described in the lookup mechanism. The second phase is retrieval itself, and is described here.

Client side

The client side has already obtained a FadaServiceMatch instance which has a method `getService()`. It will return a FadaServiceItem. This FadaServiceItem has as an attribute the service proxy requested. Calling the method `getProxy` will return a net.fada.directory.SignedMarshalledObject instance. This net.fada.directory.SignedMarshalledObject instance already contains the serialized data of the service proxy. Calling the `get()` method on this net.fada.directory.SignedMarshalledObject will start the downloading procedure, in which the class of the object whose serialized data is contained in the net.fada.directory.SignedMarshalledObject will be downloaded from the location specified by the service provider at registration time through the use of the `java.rmi.server.codebase` property.

If the client uses the FadaHelper class all this process will be executed by this class, and the client side will cleanly obtain an Object instance.

No matter what method is used, the Object instance must be assigned and cast onto a variable of type the interface that was searched. In this way the methods provided by the interface (and implemented by the class downloaded) can be called from the client, and the service proxy (and ultimately the back-end service) will be used.

Server side

The server side is divided in several parties:

– The FADA node: When the method getService() is called in the FadaServiceMatch instance obtained by the client side, this method will send a request to the FADA node. This request tells the FADA node to take the FadaServiceItem instance provided by the service provider at registration time, serialize it and return it to the caller. In this moment the job of the FADA has been accomplished, and will take no further actions.

– The http server on the service provider side: It is providing the contents of a directory or a jar file whose url has been set as the java.rmi.server.codebase property by the service provider at the time it registered the service proxy in the FADA. When the instance of net.fada.directory.SignedMarshalledObject is in the client's JVM and it calls the get() method, a class loader in the client's JVM will open that url and obtain the .class file that describes the service proxy. If the `java.rmi.server.codebase` property specified a directory (denoted by a trailing slash), only that .class file will be sent to the client (and provided by the http server on the service provider side). If the `java.rmi.server.codebase` property specified a .jar file the whole .jar file will be sent. If the service proxy only needs its own class to work, the first approach saves time and bandwidth. If the service proxy needs the aid of many classes that are not standard and implemented in every JVM, it is better to embed them all in a jar file, because all of them classes will be downloaded at once. As the jar file is compressed, the downloading time is potentially less. The role of the http server is to provide the .class or .jar file requested by an HTTP GET request issued by the class loader in the client's JVM.

– The back-end service: Unless the service proxy is a self-contained application, it will need to interact with a database, or to perform some computation too expensive for the client's JVM. The service proxy must then communicate in some way (not specified nor constrained by the FADA) with the back-end service. A typical example is a service proxy that provides a flight reservation service: the service proxy only sends the back-end service the intention of the user to reserve a flight ticket, but is the back-end service the one who resolves the concurrent access problem, checks the availability of such ticket, charges the customer, notifies the company, updates the database, etc.

The FADA stub - skeleton compiler

A client-server application will be formed from two software entities executing in potentially separated machines. The client has to request the server to perform certain operations, and the server must perform them and return the results to the client. The business logic of these operations is dependent on every application. The communication between the client and the server, however, is not part of the business logic, but must be dealt with and programmed on both sides of the application. More than that, that layer of the application can be standardized in a way that makes it easy to plug different business logic implementations. This is what Sun Microsystems did in the eighties when they developed the Remote Procedure Call (RPC) framework, and again with the Java Remote Method Invocation (RMI).

However, noting that the RMI framework is not ideal in all cases, during the development of the FADA project it has been decided to provide yet another remote method invocation mechanism. In contrast with RMI, the transport layer can be changed easily, and can be therefore adapted to a particular application needs.

In a remote call it is needed to transfer the call parameters to the server side, and once the call has been performed, to return the results back to the caller. In RPC this was accomplished by the means of a standardized data representation format, the eXternal Data Representation (XDR) format. In Java RMI, parameters and return types are simply serialized, because there is no need to provide a machine independent data format: on both sides of the call there are data compatible Java Virtual Machine instances.

Within the FADA project it was decided to keep this simple approach due to its inherently greater performance, because data is just converted to a stream of bytes, rather than packed in data containers that add an increased level of complexity.

The details of implementation and usage resemble those from the RMI framework, but differ in some aspects:

- A default transport layer, consisting in a stream of bytes sent over HTTP, is provided. The use of HTTP as the transport protocol allows to use easily available HTTP proxies or gateways, without the need to provide wrappers on the server side.

- The transport layer can be changed to whatever the service developer feels more comfortable with, or depending on the deployment scenario, whatever fits best the conditions present. The binary over HTTP protocol is only given as a default, and can easily be changed.

A developer wishing to take advantage of the FADA stub-skeleton compiler must follow a short list of simple rules:

- The server-side class must implement an interface that extends the interface net.fada.remote.Remote.

- The server-side class must extend the class net.fada.remote.RemoteObject.

- The remote methods on the interface are tagged by stating that they throw the exception net.fada.remote.RemoteException in the throws clause.

- The parameter and return types of the method calls must refer to classes that will be present on both the server and client classpath. The use of interfaces present in the classpath to represent classes that are not in the class path is allowed. However, this practice forces the client to provide such classes in an HTTP server, what is not always easy to accomplish. Therefore this practice is discouraged unless absolutely needed.

- The server-side class must export itself by calling the method export inherited from the class net.fada.remote.RemoteObject. This method needs two parameters. The first parameter is an instance of a class that implements the interface net.fada.transport.ServerTransport. A default implementation is provided by the class net.fada.transport.ServerTransportImpl. The second parameter is an instance of a class that implements the interface net.fada.transport.ClientTransport. A default implementation is provided by the class net.fada.transport.ClientTransportImpl.

- The result of the invocation of the method export is an instance of a class that implements the interface net.fada.remote.Remote. This class must be made available to the client of the service. The method encouraged within the FADA framework is to embed that class in the proxy for the service, or even to provide it as the proxy itself, as long as the server object implements the interface that will be used to register the proxy in the FADA.

- The means by which the resulting object is made available to the client is not constrained to use the FADA. However, it is an easy way to accomplish that, and therefore it is encouraged to follow the practice, either by registering the object itself or by embedding the object in an instance of the service proxy.

After the server class has been developed, the stub and the skeleton must be created. For that, the FADA development teams provides the FADA stub – skeleton compiler. It comes bundled in a jar file called fadagen.jar, a pun on the name of the stub/skeleton generator of the RPC framework, rpcgen. The invocation of the compiler has the following appearance:

```
java -jar [-classpath <additional class path>] fadagen.jar
[-classpath <additional class path>] <server class name>
```

For example, if the server class name is Server the invocation would be something similar to this:

```
java -jar fadagen.jar Server
```

In the example no additional class path were given. In case they are specified, the first classpath is used to provide the compiler with any classes used by the server that may be needed for successful generation of the stub and the skeleton. The second classpath is used to provide the javac compiler with any classes used by the server that may be needed for successful compilation of the generated stub and skeleton. The result of the invocation (in absence of errors) is a pair of new class files, named <server class name>_Stub.class and <server class name>_Skel.class, where <server class name> is substituted by the class name of the server class, provided in the above command line. For example, if the server class name is Server.class, two new files, Server_Stub.class and Server_Skel.class will be created.

The obtained stub must be made available to the client of the service. The usual way to accomplish this is to put the stub and all needed classes in a jar file that is used as the codebase annotation of a java.rmi.MarshalledObject that is then sent to the client. The FADA uses exactly this mechanism to deliver the service proxies.

Use of different communication protocols

The default communication protocol implementation provided by the classes ServerTransportImpl and ClientTransportImpl use HTTP as

the transport protocol to send binary messages from the client to the server and back. However, one may decide that plain HTTP is suitable for his particular case. For example, a private medium may be chosen. Or a protocol that ensures integrity of the message. Or maybe something completely unrelated to TCP, if both the server and the client have the capabilities to use a different type of network.

To use a different communication protocol the developer must implement three interfaces in the package net.fada.transport: ServerTransport, ClientTransport and EndPoint. Let's see how they look like:

```
package net.fada.transport;

import java.net.*;

public interface ClientTransport {

    public void setUrl( String url )
    throws MalformedURLException;

    public byte[] callMethod( byte[] request )
    throws net.fada.remote.RemoteException;

}
```

The interface ClientTransport defines two methods, setUrl and callMethod. The first method, callUrl, is used by the RemoteObject export method to make the implementation of the ClientTransport class aware of the location in the network of the server side, that is, the ServerTransport implementation class.

The second method, callMethod, takes as a parameter an array of bytes that contain the binary representation of the call generated by the stub class. This binary representation must be made available to the ServerTransport implementation class whose URL has been notified by the RemoteObject.export method via the call to the setUrl method explained before.

The ServerTransport interface is the following:

```
package net.fada.transport;

public interface ServerTransport {

    public net.fada.transport.EndPoint receiveCall()
         throws net.fada.remote.RemoteException,
          java.io.IOException;
    public void start() throws java.io.IOException;
```

```
}
```

This interface defines two methods, start and receiveCall. The first method, start, is used by the RemoteObject instance to signal the ServerTransport implementation class that it must start listening for incoming requests from the client side, that is, the ClientTransport implementation class.

The second method, receiveCall, must be made a blocking call. It will be called by the skeleton class when it is ready to receive incoming requests. When a request is sent from the client side (that is, the ClientTransport implementation class) and received by the ServerTransport implementation class the receiveCall method must return from the call, returning an implementation of the interface EndPoint. The skeleton will then use the EndPoint implementation class to serve the request. The receiveCall method must not make any effort to receive concurrent requests: the skeleton class takes care of that.

The EndPoint interface is like this:

```
package net.fada.transport;

public interface EndPoint {

    public byte[] getBytes();

    public void send( byte[] bytes ) throws java.io.IOException;

    public void close();
}
```

The EndPoint interface has three methods: getBytes, send and close. The first method, getBytes, obtains the received binary call representation. This method is called by the receiver thread in the skeleton.

The method send is used by the skeleton to send back the binary response to the client (the ClientTransport implementation class that is being used by the stub class).

The method close is used by the skeleton to force the release of any resources in use by the instance of the class EndPoint, such as sockets or Input/OutputStreams.

A description of the communication scenario is as follows: the export method on the RemoteObject has been called. The RemoteObject receives the instances of the implementations of the interfaces ServerTransport and ClientTransport, plus the URL where the server (skeleton) will be visible. The RemoteObject calls the setUrl method on the ClientTransport implementation class. Then the ClientTransport implementation class is made available to the stub. Then the start method is called on the ServerTransport implementation class. The ServerTransport implementation class starts listening for incoming requests. Then the export method on RemoteObject returns the stub. The stub must be made then available to the client side. Note that the stub class implements the interface serializable, and will be transferred to the client side. Therefore, the enclosed ClientTransport implementation class must also be made serializable.

The client receives the stub class. A way to do this is by enclosing the stub in an instance of a service proxy. That's the way FADA does it for its own stubs, and the recommended way for developers of service proxies that use FADA.

When the client calls one of the stub methods (directly or via a call to a method in the service proxy) it prepares the binary representation of the method call, and uses the ClientTransport implementation class callMethod method to send it to the server side (that is, the ServerTransport implementation class). The ServerTransport implementation class is blocked waiting for incoming requests in the method receiveCall. When the request is received the receiveCall method unblocks and returns the EndPoint instance to the skeleton. There is no need to make the EndPoint class serializable, as it won't ever travel from the client to the server.

The skeleton then services the request by calling the appropiate method in the RemoteObject class. That method may return a return value. This return value is converted to a binary chunk of data that is sent to the client side via the send method in the implementation class of the interface EndPoint, constructed by the ServerTransport implementation class method receiveCall. The binary data chunk then arrives at the client side, the method callMethod in the ClientTransport implementation class. This method returns the binary data to the stub class, which then opens the binary data chunk and retrieves the return types (or exceptions, if it is the case) and returns them to the caller, the service proxy or the client application.

To implement a different communication protocol, the three interfaces must be implemented. It amounts to a total of seven

methods, the majority of which are not complicated at all. The meat is mainly in the callMethod and receiveCall methods, which encode and decode the binary data chunk in messages that can be transferred by using the communication protocol of choice, be it HTTP (as in the default implementations), HTTPS, SSL, or a brand new encrypted-integrity-checked-non-repudiable communication protocol.

Multicast extensions

Althogh FADA was born to work properly in a WAN environment, its use can be extended to LAN environments working in a similar manner than JINI Network™ does it.

For this reason, certain extensions has been added to the functionality provided by a FADA node when it works in a LAN environment or other kind of networks where communications over multicast protocols/addresses are allowed.

There are two extensions added. Both are related to event management, but one of them is the base for the other one:

- Events: Allows to send and receive events related to facts occurred in the FADA nodes.

- Discovery: Allows to discover all the active nodes present in the same LAN where discover agent performs the discovery.

In the actual implementation of these multicast extensions, the multicast address and port used to send and receive the UDP packets involved when discovery mechanisms or event notifications are performed is specified in the table below:

Multicast address	224.200.200.244
Multicast port	4004

NOTE: *These constants are defined in the JAVA class* `net.fada.Constants`*. If you need to change the value of this constants for any reason, change the values of these contants specified inside quoted class and recompile the FADA source to apply properly the changes.*

Multicast events

Multicast events implemented in FADA will only work porperly in LAN environments due a multicast protocol is used for sending and receiving UDP packets containing the events. We discard in purpose the WAN environments (Internet particularly) as multicast allowed

medium due most of the routers that mesh this kind of network are not able to support routing protocols for multicast addresses.

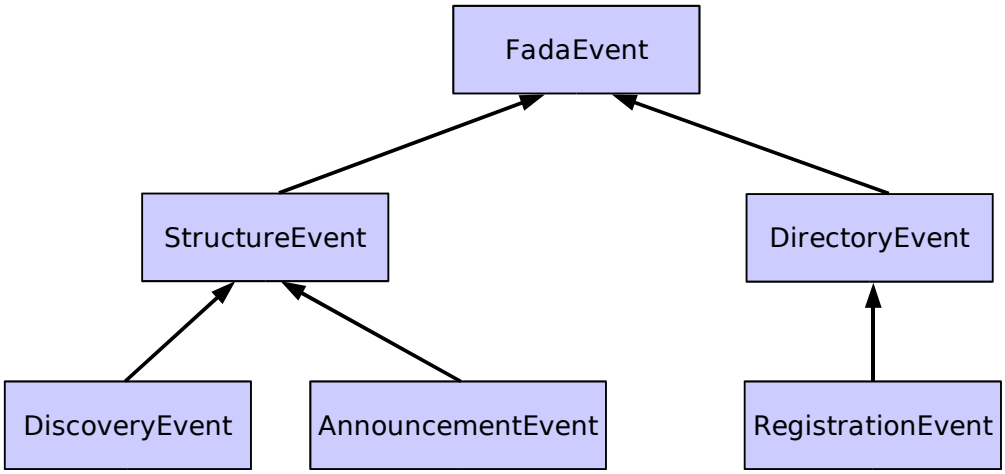
This event management mechanisms have been implemented over a unreliable multicast protocol.

The events are sent to the network using UDP packets containing the JAVA object representing the event in a serialized format. It implies that the event enabled receivers must have in its CLASSPATH the JAVA classes representing these events for a correct desearialization and re-instantiation in its own JVM.

In the next chapter, it is shown the event hierarchy designed for the FADA event management.

Events hierarchy

The event hierarchy implemented for FADA event management is shown in the next picture:



The parent class for all events present in the FADA event management is called FadaEvent. From this agnostic event representation class that encloses all the FADA related events hangs two subtypes from it:

- StructureEvent: This kind of event and all of its subimplementations are intended to represent events related to

facts occurred and enclosed to the FADA structural functionality field. In other words, the events representing all the stuff related to the structure of the FADA platform as discoveries or announcements (useful for create links between nearby nodes).

The subimplementations defined for this event are:

- DiscoveryEvent: This event is used to perform the discovery of the FADA nodes present in the same LAN. This event will be accepted and processed by the FADA nodes enabled to dispatch multicast events. This kind of event is useful for a FADA-enabled applications to discover nodes avoiding to have any customized configuration for searching nodes.

- AnnouncementEvent: This event is used to perform the discovery of the FADA nodes present in the same LAN, and create a neighborhood links between FADA nodes. This event will be accepted and processed by the FADA nodes enabled to dispatch multicast events. This kind of event is ONLY useful for the FADA node's application. The main goal for using this event by FADA nodes is for meshing (in an automagically way) the FADA network present in the same LAN.

- DirectoryEvent: This kind of event and all of its subimplementations enclose the representation of all events related to the FADA directory functionality field.

The subimplementations defined for this event are:

- RegistrationEvent: This event is used to notify that something was happens about the registration of a FADA service proxy or a set of FADA service proxies with a specific characteristics (FadaServiceTemplate). This event is always generated and sent by FADA nodes and received and processed by all FADA-enabled applications that using the FadaHelper utility class. The applications which are interested in intercept these events must be registered as listener for this kind of events in the FadaHelper used by them.

Multicast discovery

The multicast discovery mechanism is truly useful when the FADA technology is used by a software solution completely placed in a LAN environment. The mechanism is easy to understand and is explained in the following points:

- Discoverer agent sends a DiscoveryEvent to the network. This DiscoveryEvent is created passing to the constructor an AnnouncementPacket object in which is specified the information about the location of the discoverer agent who sends the event.

- Discoverer agent actives a process to receive and dispatch all the responses to the DiscoveryEvent by the multicast-enabled FADA nodes presents in its same LAN.

- Discovered nodes sends its own location information as response to DiscoveryEvent sent by discoverer agent. This location information is useful for creating a FadaLookupLocator pointing to it by the discoverer agent aware application. The Java object that represents this information is the `net.fada.directory.tool.FadaNodeInfo`.

- Discoverer agent process receives the response sent by the nodes and dispatch the information to all the listeners registered which are interested in the FADA node discovery.

The utility class to perform the discovery mechanism by the FADA enabled applications is called `net.fada.toolkit.FadaDiscovery`. In later chapters, an example shows how to use properly this class to discover FADA nodes using the multicast discovery mechanism.

NOTE: *Previous implementation of a discovery agent helper can still be found in the `net.fada.toolkit.FadaHelper` ("`discovery`" method) but IS COMPLETELY DEPRECATED AND IT MAY NOT BE USED.*

Multicast Announcement

The multicast announcement mechanism is very similar to the multicast discovery mechanism explained in the previous chapter, but the main difference is the target components which will use it. In this case, this mechanism is ONLY useful by the multicast enabled FADA nodes. The objective of this mechanism is not only to discover nearby FADA nodes but linking both as neighbors. This mechanism permits to mesh the FADA network in a transparent manner.

The mechanism is performed according to the steps explained in below points:

- Announcer node sends an AnnouncementEvent to the network in a serialized format embedded in a UDP packet. This AnnouncementEvent contains an AnnouncementPacket with the location information of the announcer node.

- Multicast enabled FADA nodes in response to the AnnouncementEvent received performs a connection with the announcer node establishing with it a neighborhood bidirectional relationship. The mechanism to establish this relationship is the same that is used when FADA node administrator manually connects nodes using web based management or lookup originator receives a lookup result from other FADA node performed during the extension lookup mechanism.

For now, this mechanism is not invokable from web based management or, in other words, manually executed and is only performed when a multicast enabled FADA node is starting up.

Registration events

In some cases can be useful to FADA enabled applications to be aware that some event related to a particular kind of service or services have occurred. For instance, it may be interesting that an application (acting as FADA service proxy consumer) is notified when a particular service has been registered or deregistered from the FADA network (action performed by the FADA service proxy provider).

The RegistrationEvent is a JAVA object that contains the embedded information:

- All useful information about the FADA service proxy involved in the registration action. This information is wrapped in a only class (`net.fada.directory.tool.FadaServiceMatch`).

- Type of the registration action has occurred. This type is characterized with an integer. There are three types:

  - NOMATCH_MATCH: integer value is 0

  - MATCH_MATCH: integer value is 1

  - MATCH_NOMATCH: integer value is 2

A multicast enabled FADA node sends a new RegistrationEvent to the network in response to certain actions taken place in it and related to the registration of a service:

- Service has been **registered**: Node sends a RegistrationEvent specifying the type of the event as a NOMATCH_MATCH RegistrationEvent.

- Service has been **re-registered**: Node sends a RegistrationEvent specifying the type of the event as a MATCH_MATCH RegistrationEvent.

- Service has been **deregistered**: Node sends a RegistrationEvent specifying the type of the event as a MATCH_NOMATCH RegistrationEvent.

As we can see all the events occurred in each multicast enabled FADA node will be sent to the network ignoring if they are consumed or not.

The way to consume this events by the FADA enabled applications is using the utility class `net.fada.toolkit.FadaHelper`. (the same class that is used to simplify the FADA service proxies registration and its renewal in a FADA node). In this case, the application which wants to be notified when RegistrationEvents related to a particular kind of service occur MUST implemented the interface `net.fada.directory.tool.RegistrationEventListener`. This implementation will be registered as listener for the events occurred according to the kind of the service specified by the `net.fada.directory.tool.FadaServiceTemplate` passed as parameter during the listener registration phase.

To register the listener is so easy as calling to the `FadaHelper`'s method called `addRegistrationEventListener` passing as the first parameter the listener and as the second one, an instance of `net.fada.directory.tool.FadaServiceTemplate` which specifies in what kind of service proxies is interested to consume its related registration events.

In later chapters, it is shown an example demonstrating how to enable a FADA enabled application to be aware of RegistrationEvents.

Code Examples

One of the main criticisms about the Jini Networking Technology is that its Application Program Interface (API) is not straightforward to understand and use. Having used it, and having to agree with this opinion, the developers of the FADA decided to simplify the FADA API without sacrificing flexibility and control over the execution.

Although it is possible to use the FADA API classes in a Jini style, that is, going down to the lowest level to build an application from the basic building blocks, the primary way to use the FADA is through a helper class, called consequently FadaHelper, that makes it easy to perform the most common operations with the FADA.

This class has methods for registration of services and to search for interface implementations. The class method has parameters and return types that may require the invocation of lower-level classes and methods, but not to the extent to make it cryptical.

NOTE: *An additional document called "FADA Tutorial" is being written to explain with more detail all the posibilities of use of the FADA API and all its extensions. A draft release of this document can be retrieved from the official FADA site.*

Discovering nodes using multicast discovery

Before to register a service proxy in a FADA network is needed to access to one of the nodes presents in it. In case of the FADA network is completely place in a LAN environment, the multicast discovery can be used by FADA enabled applications to find one or more FADA nodes. To perform this multicast discovery is used the utility class `net.fada.toolkit.FadaDiscovery`. But for using this class before is needed to develop a class that implements the interface `net.fada.toolkit.DiscoveryListener`. For instance:

```
public class MyDiscoveryListener implements DiscoveryListener{
  public void discovered ( FadaLookupLocator locator ){
    System.out.println( "Fada Node discovered at:" +
      locator.toString() );
  }
}
```

Now this class is registered as discovery listener in a instance of a FadaDiscovery class. This listener will be notified for each response from the FADA node present in the LAN. The discovery mechanism starts when the listener is registered in the FadaDiscovery instance sending to the network a DiscoveryEvent. In next paragraph, it is shown a little code snippet that uses the FadaDiscovery utility class to perform the multicast discovery:

```
public static void main( String[] args ){

  // constructs the customized discovery events listener
  MyDiscoveryListener mdl = new MyDiscoveryListener();

  // creates an instance of discovery agent
  FadaDiscovery discoverAgent = new FadaDiscovery();

  // registers the listener and (in transparent way to the
  // programmer) the DiscoveryEvent is sent to the network by
  // the agent
  discoverAgent.addDiscoveryListener( mdl );

  // until “ENTER” key is not presset the program is waiting for
  // receiving discovery responses from the FADA nodes
  System.out.println( “PRESS ENTER TO INTERRUPT THE
DISCOVERY” );
  System.in.read();
}
```

Registering as RegistrationEvent listener

In similar way to explained in the previous chapter, to consume the RegistrationEvents produced by multicast enabled FADA nodes, a class that implements the interface `net.fada.directory.tool.RegistrationEventListener` must be developed. This class will be notified when RegistrationEvents for a certain kind of service proxies occurs. For instance:

```
public class MyRegistrationEventListener implements
net.fada.directory.tool.RegistrationEventListener {

  public MyRegistrationEventListener(){
    // empty constructor
```

```
  }
  public void serviceRegistered( RegistrationEvent ev ){
    FadaServiceMatch match = ev.getSource();
    System.out.println( “Service Proxy with ID: “ +
match.getFadaServiceID() + “ has been registered!! );
  }
  public void serviceDeregistered( RegistrationEvent ev ){
    FadaServiceMatch match = ev.getSource();
    System.out.println( “Service Proxy with ID: “ +
match.getFadaServiceID() + “ has been deregistered!! );
  }
  public void serviceReregistered( RegistrationEvent ev ){
    FadaServiceMatch match = ev.getSource();
    System.out.println( “Service Proxy with ID: “ +
match.getFadaServiceID() + “ has been reregistered!! );
  }
}
```

The following code snippet shows how register the listener in the FadaHelper utility class.

```
public static void main( String[] args ){
  // creates the listener
  MyRegistrationEventListener listener = new
MyRegistrationEventListener();

  // creates an instance of FadaServiceTemplate to specify in
  // which service proxy's related registration events the
  // application is interested
  FadaServiceTemplate template =
      new FadaServiceMatch(
        new String[ “entry1” ],
        null,
        new String[ “net.fada.examples.service1Interface”]
      );
  // creates the FadaHelper ( our RegistrationEvent dispatcher )
  FadaHelper helper = new FadaHelper( new FadaLeaseRenewer() );

  // registers the listener and the template
  // from here on listener will receive all the
```

```
    // RegistrationEvents that matching with the template
    helper.addRegistrationEventListener( listener, template );

    ...

    // deregisters the listener to give up receiving notifications
    // about RegistrationEvents
    helper.remoteRegistrationEventListener( listener );
}
```

Registering a Service

In order to make a service proxy available to the FADA federation, it must be registered on a FADA node. The FadaHelper method to perform this is the register method, which has the following signature:

```
public FadaServiceID register(
  FadaInterface fp,
  java.io.Serializable item,
  FadaServiceID id,
  String[] entries,
  long leasePeriod,
  net.fada.directory.security.SecurityWrapper wrapper,
  String annotation,
  RenewalEventListener listener)
throws FadaException,
    IOException,
    NullPointerException,
    java.security.InvalidKeyException
```

The first parameter is the FADA proxy of an active FADA node where the service proxy is to be registered (this proxy can be obtained using the `net.fada.directory.FadaLookupLocator`). Note that no matter where the service proxy is registered, if the FADA node belongs to a federation of FADA nodes, the registered service proxy will be accessible to all members of that federation.

The second parameter is the service proxy itself. It must be a serializable object, otherwise it won't be able to travel from the server to the FADA node, and from the FADA node to the requesting

client. It must implement at least one interface that is known to one or more clients.

The third parameter is the FADA service identifier related to the service proxy which will be registered. In case of it is the first registration of the service proxy this parameter must setup to null value. It this method is called to perform a reregistration, the value of this parameter must be equals to the returning value obtained in the first registration.

The fourth parameter is an array of Strings that contain qualifying properties of the service proxy. For example, if a service proxy is related to hotels, the word “Hotel” could be part of the entry set. This allows clients to fine search the FADA federation, or to broaden it, if they prefer. More on this will be seen on the section on searching or looking up proxies.

The fifth is a long, containing the number of milliseconds the server wants the service proxy to stay registered as minimum. This quantity will be the initial lease time for the registration of the proxy, provided that it is accepted by the contacted FADA node.

The sixth parameter is the securityWrapper object used to apply the security mechanisms which signs the service proxy registered allowing to potencial service proxy consumers to check the identity and integrity of this when they performs the lookup mechanisms. More about security mechanisms applied to consuming FADA service proxies is detailed at chapter [FADA security wrappers]. If it is not wanted to apply this security constrains to the registration of the service proxy, the value of this parameter must be setup to null value.

The seventh parameter is the codebase annotation where client consumers of this service proxy can download the needed classes to recreate the service proxy in its own JVM after lookup phase. The codebase annotation can be setup in two ways: using this parameter or using the `java.rmi.server.codebase` JVM system property. We encourage to use the first way due it gives more flexibility.

The eighth parameter and the last is the object that implements the interface net.fada.directory.tool.RenewalEventListener which will be notified in case of the underlaying process in charge of renewing the lease of the service proxy registered is unable to perform the renewal.

The result of the call is a FadaServiceID. Every registered proxy obtains a globally unique identifier. This identifier can be later used with some other methods of the FadaHelper.

You may have noticed that the FadaHelper class register method is not marked static. It means that you need to create an instance of the class in order to use the registration feature. The constructor is as follows:

```
public net.fada.toolkit.FadaHelper(
  net.fada.directory.tool.FadaLeaseRenewer )
throws java.lang.NullPointerException;
```

The constructor takes as a parameter an instance of a class called FadaLeaseRenewer. As its own names states, this class will be in charge of automatically renewing the service proxy lease in behalf of the server. By making the registration method non-static the developer is forced to create an instance of both the FadaHelper and the FadaLeaseRenewer. These instances should be created from within the server code. In this way, should a server crash occur, the registration lease would be destroyed with it, and it would eventually expire.

The FadaLeaseRenewer constructor is as follows:

```
public net.fada.directory.tool.FadaLeaseRenewer();
```

As you can see, it takes no parameters. Upon registration of service proxies the FadaHelper notifies the FadaLeaseRenewer and the latter keeps renewing the registration leases indefinitely.

The registration method works as follows:

- It contacts the FADA node whose FADA proxy was given as a parameter.
- It wraps the service proxy in an instance of `net.fada.directory.SignedMarshalledObject`.
- It constructs an instance of `net.fada.directory.FadaServiceItem` that wraps the service proxy passed as a parameter, together with the entries provided. The service proxy is not sent as-is, but rather wrapped in an instance of java.rmi.MarshalledObject.

- It uses the FADA proxy methods to notify the FADA node of the intention to register a service proxy. The FADA proxy method sends the `FadaServiceItem` instance and requests a lease time as specified by the parameter leaseTime.

- If the FADA node accepts the specified lease time, the proxy, together with the information wrapped in the FadaServiceItem instance, is stored in the FADA node, and it returns an instance of a class that digests all the registration information.

- This registration information is stored by the FadaHelper, which then only returns the FadaServiceID to the caller of the method register.

The method register in the class FadaHelper is overloaded to allow different registration scenarios.

How does the net.fada.directory.SignedMarshalledObject work?

In summary, `net.fada.directory.SignedMarshalledObject` is a reimplementation of `java.rmi.MarchalledObject` in which has been added extra security mechanisms. For further information about the security mechanisms added to this class see the chapter [FADA security wrappers]. By the way, analogous to its predecessor, `net.fada.directory.SignedMarshalledObject` is an instance of a is a serializable object that contains a serialized instance of a serializable object within, and keeps an annotation with it. This annotation denotes the resource where the .class files for the wrapped serializable object can be found. The `net.fada.directory.SignedMarshalledObject` has no magical ways to know where those class files can be found. Instead, it relies in the property *java.rmi.server.codebase* or setted up by a parameter passed in the constructor for this object. In order for a SignedMarshalledObject to work properly this property or parameter must be set with an appropriate value. If, for instance, a SignedMarshalledObject is constructed with an instance of the class Foo, the codebase annotation must point to a resource where the file Foo.class (and associated classes) can be found. For example, if the URL *http://myserver.mydomain.com* contains an HTTP server running, and the path `/classes/myJar.jar` contains all classes needed, then a valid codebase annotation can be `http://myserver.mydomain.com/classes/myJar.jar`.

The SignedMarshalledObject will be sooner or later serialized and sent to a client. The client will deserialize the instance of the SignedMarshalledObject. When he does so, the wrapped class is still stored in serialized format within the instance of the SignedMarshalledObject. If the client then calls the SignedMarshalledObject's `get()` method, it will attempt to deserialize the class contained in the SignedMarshalledObject. In doing so it will need the classes used by the serialized instance. To download the classes it will open the URL provided as the annotation. It is the responsibility of the creator of the instance of SignedMarshalledObject to provide a correct codebase annotation. There is nothing the client can do if it obtains an instance of a SignedMarshalledObject whose codebase annotation points nowhere, an invalid URL, or a URL that doesn't contain the expected classes. In all of these cases a `ClassNotFoundException` will be thrown.

The FADA methods make no explicit reference (if you are using the helper classes) to the `net.fada.directory.SignedMarshalledObject` class, but they use it extensively when it comes to registering and searching the FADA architecture. Their appearances have been intentionally hidden from the API to simplify matters, but both the client and the server side must be aware of their presence, and must understand how it works, and the security hazards that may arise when foreign code is allowed to execute in a JVM instance. For more information, see the Sun's Java website (http://java.sun.com) and the information on marshalled objects.

Deregistering a service

The registration of a service proxy is not kept indefinitely by the FADA node, but only for the duration of a lease period. The FadaHelper uses the FadaLeaseRenewer instance provided at instantiation time to renew this lease indefinitely. If the server side wishes to deregister the service proxy for whatever reason, it can do so by calling the deregister method, whose signature is as follows:

```
public void deregister(
  net.fada.directory.tool.FadaServiceID
);
```

Note that the method is not static either. The FadaServiceID instances obtained upon registration of service proxies, along with more information returned by the FADA node, such as the

registration lease, is kept internally by the FadaHelper class. This information is of interest only for the registerer, the server class. By making the registration and deregistration methods non-static, and thus forcing the server side to call the constructor of the class FadaHelper and FadaLeaseRenewer, this sensitive information is kept hidden from foreign eyes and hands. No one can deregister a service proxy, except the server which registered, and the administrator of the FADA node where it was registered.

Upon calling the method deregister, the FadaHelper searches its internal registry, retrieves the information needed, and asks the FadaLeaseRenewer to cancel and stop renewing the lease associated with the service proxy whose FadaServiceID was given in the call to deregister.

A benefit of having all this information hidden in an instance of FadaHelper is that no one else knows what is the lease for the registered service proxy. In this way it is very difficult to accidentally (or intentionally) deregister a service proxy whose associated server is working without problem.

Looking up a Service

The class FadaHelper also contains a method to perform lookup of a set of service proxies in the FADA architecture. The signature of such method is:

```
public static java.lang.Object[] lookup(
  java.lang.String url,
  java.lang.String[] interfaces,
  net.fada.directory.tool.FadaServiceID id,
  java.lang.String[] entries,
  int maxResponses,
  long timeout )
throws net.fada.FadaException,
  java.io.IOException,
  java.lang.ClassNotFoundException;
```

The first parameter is the url of a FADA node. Note that, in order to find a service proxy, it is not needed to start the lookup request on the same FADA node as the one in which the service proxy was registered. The FADA architecture takes care of extending the search throughout all the neighbors of the FADA node that receives the lookup request from the client.

The second parameter is an array of strings that contains the names of the interfaces it is desired to look for. If it is wished to look for an implementation of the interface com.fetishproject.HotelSearch, for example, this parameter must contain a string with the contents "`com.fetishproject.HotelSearch`".

The third parameter is the service id of a particular service proxy. This allows to search only for a specific implementation of the interfaces. If it is known that a certain implementation of a certain set of interfaces has been registered in the FADA architecture, and its FadaServiceID is known to the client, it may request the FADA architecture to search and retrieve just this particular service proxy. Note that in this case the maximum number of proxies returned will be one, as no two service proxies may have obtained the same FadaServiceID. FadaServiceIDs are globally unique.

The fourth parameter provides a way to narrow the search. Following the HotelSearch example given above, it may be wished to select only service proxies that were registered together with the keyword "Rome", for example. In this case, the fourth parameter would contain a string with the word "Rome", and only those proxies that were registered with at least this entry would be returned.

The fifth parameter is the maximum number of responses wanted. In a FADA federation there may be an indefinite number of service proxies, and probably the client is not interested in all of them. By specifying a limit in the form of this parameter it is possible to restrict the search to a maximum number of responses. In this way the amount of memory used by the client may be controlled.

Finally, the sixth parameter is the maximum number of milliseconds the client wants to wait for responses. Knowing that the FADA architecture has an unbounded size, and that there is no way to know if all the FADA nodes have been visited (due to the lack of a central coordination entity), the client puts the limit on the time it wants to wait for results.

The result is an array of Object. These objects are the service proxies that match the query parameters. All of them implement all of the specified interfaces (if any), so you can cast them onto a variable of type the interface of interest, and call the methods in the interface.

How does the matching mechanism work?

The lookup parameters are three: the FadaServiceID, the set of interfaces and the set of entries. Once received by the FADA node, it considers a service proxy matches the lookup requirements if it has the same FadaServiceID as the one provided, and it implements all of the provided interfaces, and it has an entry set that features at least one matching entry for every provided entry. Entries are compared on a byte by byte basis.

Each and every of these parameters can be null. A null parameter is not taken in account when matching service proxies. So if you specified a null FadaServiceID only the interfaces and entries will be compared to give a match. If all of the parameters are specified as null, all the proxies registered will match the query, so it isn't a very good idea to specify all parameters as null, unless you really want all of the proxies registered in the whole FADA architecture (remember that lookup requests aren't limited to a FADA node unless explicitly stated in the lookup parameters).

The lookup method is overloaded so it is not necessary to specify null for some of the parameters. Default values will be used if any of the overloaded methods are used when searching the FADA architecture.

Complete example

In this section a complete example on the use of the FADA to distribute service proxies will be provided. A FADA enabled service consists of two parts: the server and the service proxy. An additional boot-strap part will be necessary to register the service proxy in the FADA. Note that this part should be as tightly tied to the life-cycle of the service as possible. In this way the leasing mechanism is be as effective as can be.

First of all an interface for the service must be chosen. As an example a remote printer service will be provided. As such interface does not exist it will be created in this example.

A printing service should have a method to print something. This something will be a file. Therefore, the interface will look like this:

```
import java.io.*;

public interface RemotePrinter {
  public void print( File file ) throws IOException;
}
```

It has only one method, but given the nature of the service little more can be done. The only method is called print, and accepts a java.io.File as a parameter. It is declared to throw a `java.io.IOException` in case there is a problem retrieving the file.

In order to make use of this interface the client must have a copy of it in the classpath, both at compile and at run time.

Service provider side

The service provider is responsible for both the creation of the server and the creation of the service proxy. In the first step the server will be created.

```
import java.net.*;
import java.io.*;
import net.fada.toolkit.*;
import net.fada.directory.tool.*;

public class Printer {

  int portNumber;
  ServerSocket serverSocket;
  Thread runner;

  public Printer( int portNumber ) throws IOException {
    this.portNumber = portNumber;
    this.serverSocket = new ServerSocket( portNumber );
  }

  class ListenerThread implements Runnable {
    public void run() {
      for(;;) {
        Socket clientSocket = null;
        try {
          clientSocket = serverSocket.accept();
        } catch( Exception ex ) {
          continue;
        }
        Thread serverThread = new Thread(
          new ServerThread( clientSocket )
```

```
          );
          serverThread.start();
        }
      }
    }

    class ServerThread implements Runnable {
      Socket s;
      public ServerThread( Socket s ) {
        this.s = s;
      }
      public void run() {
        InputStream is = null;
        try {
          is = s.getInputStream();
          ByteArrayOutputStream baos = new ByteArrayOutputStream
();
          int read = -1;
          byte[] chunk = new byte[ 1024 ];
          while( ( read = is.read( chunk ) ) != -1 ) {
            baos.write( chunk, 0, read );
          }
          // All bytes read, now print them
          /*
           *    The interaction with a real printer has been left
out
           *    of the example because it adds nothing to the
           *    illustrative purposes.
           */
        } catch( Exception ex ) {
        }
      }
    }

    public void start( String serverUrl, int port, String
fadaUrl )
    throws Exception {
      // Create the instance of the service proxy
      PrinterProxy pp = new PrinterProxy( serverUrl, port );
      // Prepare listener thread
      runner = new Thread( new ListenerThread() );
```

```
    // Start listening for requests
    runner.start();
    // Prepare lease renewer
    FadaLeaseRenewer flr = new FadaLeaseRenewer();
    // Prepare the FadaHelper instance
    FadaHelper fh = new FadaHelper( flr );
    // Register the service proxy in a FADA node
    FadaServiceID id = null;
    FadaLookupLocator locator = new FadaLookupLocator(fadaUrl);
    id = fh.register(
        locator.getRegistrar(),
        pp,
        null,
        new String[] {PrinterProxy},
        10000L,
        null,
        "http://server:port/where_codebase_is.jar",
        null );
  }

  public static void main( String[] args ) throws Exception {

    // Get parameters
    // First parameter is the port number
    int portNumber = Integer.parseInt( args[0] );
    // Second parameter is the server host name, needed for the
proxy
    String url = args[1];
    // Third parameter is the url of the FADA node
    // Discovery mechanisms could be used here
    String fadaUrl = args[2];
    // Create the instance of the server
    Printer me = new Printer( portNumber );
    // Start the server
    me.start( url, portNumber, fadaUrl );
  }

}
```

Note that the server need not implement any interface. The client will only see the service proxy, so the server class can decline to implement the created interface. In fact, the server class needn't even be coded in the Java programming language. Only the service proxy needs to. But for the sake of clarity, the server will also be coded in Java.

The constructor accepts a port number, and attempts the creation of an instance of the class `java.net.ServerSocket`. If the creation of the ServerSocket fails (for example, because the port is already used) the constructor throws an `java.io.IOException` and fails.

As the server must be able to serve multiple requests, a thread is spawned for each request. Each thread will collect the data sent from the client and then send it to the printer. More efficient ways for data collection can be used, but the point in this example is to illustrate the use of the FADA architecture.

This particular server won't print anything at all, as the interface with the real printer has not been defined. However, that's all this implementation is missing. If you want you can write some information to standard output when the file is received, or you can interact with a real printer. This code is just an example of how to develop the server and the proxy, and how to use service proxies from clients.

The service proxy must also be developed by the service provider. As the communications protocol used between the service proxy and the server is only of interest for the service provider, it's up to him to decide what to use. In this example a simple byte stream has been used, with no control fields. The basic assumption is that the printer can understand any type of binary stream data, or that both the server and client sides agree that only one kind of data (for example, text files) will be ever sent. All these agreements must be known in advance by both the server and the client, and the source of such agreements can be the same source as the entity hosting the interface. If this is an impromptu service (like in the example), both the client and the service already know the restrictions and implications.

The code for the service proxy is the following:

```
import java.io.*;
import java.net.*;
```

```
public class PrinterProxy implements RemotePrinter, Serializable
{

  String host;
  int portNumber;

  public PrinterProxy() {
    // For serialization purposes
  }

  public PrinterProxy( String host, int portNumber ) {
    this.host = host;
    this.portNumber = portNumber;
  }

  public void print( File file ) throws IOException {
    // Prepare to read the file
    FileInputStream fis = new FileInputStream( file );
    // Open connection with the server
    Socket s = new Socket( this.host, this.portNumber );
    // Get the channel for writing
    OutputStream os = s.getOutputStream();
    // Read the file, and write to the server
    byte[] chunk = new byte[ 1024 ];
    int read = -1;
    while( ( read = fis.read( chunk ) ) != -1 ) {
      os.write( chunk, 0, read );
    }
    fis.close();
    os.close();
    s.close();
  }

}
```

The service proxy is the link between the client application and the server. It will be delivered to the client wrapped by an instance of a `net.fada.directory.SignedMarshalledObject`. As such, the embedded object (the service proxy) must be converted to a stream of bytes prior to be sent within a MarshalledObject. The service

proxy must be serialized. An object can be serialized if it implements the interface `java.io.Serializable` and it has a no-argument constructor. Therefore the service proxy implements the interface `java.io.Serializable` and defines a no-argument constructor, with remarks stating why it is needed. Apart from this interface, the service proxy also implements the interface known to the client, and the only knowledge the client has about the service proxy: that it implements a known interface. When the client gets the service proxy it has no way to know what class it belongs to, whether that class is in its classpath, or any other detail about the implementation. But by knowing for sure that implements the interface RemotePrinter the client can blindly cast the service proxy instance onto a variable of type RemotePrinter, and use the interface's methods. The class the service proxy belongs to has been downloaded by the `net.fada.directory.SignedMarshalledObject` instance when it was requested to return the embedded object instance. That's why the instance of the class `net.fada.directory.SignedMarshalledObject` needs the codebase annotation: to download the remote code, and that's why the codebase property must be set on the server side: because the client doesn't know where it comes from.

Client side

The client on this example will be a program that will open an file and send it to the remote printer. It must locate a FADA node, request it to search for the appropriate service proxy, and use it to perform the operation.

```
import net.fada.toolkit.*;
import java.io.*;

public class PrinterClient {

  public static void main( String[] args ) throws Exception {

    // Get the parameters
    // First parameter is the name of the file to print
    String fileName = args[0];
    // Second parameter is the url of the FADA node to contact
    String fadaUrl = args[1];

    // Declare a variable to cast the service proxy onto
    RemotePrinter service = null;
```

```
      // Prepare the lookup request parameters
      String[] interfaces = new String[] { “RemotePrinter” };

      // Perform the lookup procedure
      Object[] proxies = FadaHelper.lookup(
        fadaUrl, new String[]{ “PrinterProxy” }, null,
interfaces, 1, 10000L
      );

      // Take one of the returned service proxies
      service = (RemotePrinter) proxies[0];

      // Invoke the service method
      service.print( new File( fileName ) );
    }

}
```

This simple client will perform all its business logic in the `main()` method. It simply takes two arguments: the name of the file to print, and the url of a FADA node. As in the server side code, discovery methods can be used. But for the example it will be assumed that the client already knows the url of a FADA node.

Note that the FADA node whose url is given as a parameter need not be the same FADA node as the one used in the registration of the service proxy. It is sufficient that the FADA node contacted when performing the lookup is connected to the FADA node that contains the registered service proxy. This connection need not be direct. As long as both FADA nodes belong to the same federation they will be able to see each other.

Notes about JAVA sandbox and dynamic code loading

Marshalled Objects work on the basis of downloadable code. That means that the actual implementation of a class is downloaded from somewhere to the JVM. The execution of foreign code raises a horde of potentially hazardous situations. The foreign code could, for example, decide to delete all of your home directory, or start a DOS (Denial of Service) attack taking your machine as the origin attacker.

For these and other reasons, downloaded code is specially scrutinized in search of potentially hazardous instructions (as well as bytecode

checked upon download). A security manager makes sure that downloaded code is not allowed to do any operation it hasn't been explicitly allowed to.

The user of downloadable code specifies the set of allowed operations by the means of a policy file. The policy file contains the policies applicable to downloaded code. The security manager is the mechanisms by which these policies take effect.

Both the server and the client use downloadable code. The server first contacts a FADA node and downloads the proxy for the FADA. This code was not in the classpath of the server, and therefore it is downloaded code. Therefore, the server needs a security manager and a policy file, because otherwise the JVM refuses to execute downloaded code.

The client uses two pieces of downloadable code. It first contacts a FADA node and downloads its FADA proxy. It then uses the FADA proxy to locate and download the service proxy. Finally, it uses the service proxy. Neither the code for the FADA proxy nor the code for the service proxy were in the classpath of the client, and therefore they are downloaded from the origin site.

This all means that the users of the FADA, as well as the users of service proxies must perform two steps before attempting a successful completion of downloadable code execution:

- (Optional) Set up a security manager. The Java API provides a suitable one in the form of the class `java.rmi.RMISecurityManager`. NOTE: Since FADA (in its version 5.0.0 and above) uses `net.fada.directory.SignedMarshalledObject` as replacement of `java.rmi.MarshalledObject`d it is not needed to set explicitly the security manager, because the default security manager is used instead of.

- Provide a policy file. This policy file should contain the minimum set of permissions necessary for the execution of the downloaded code.

The policy file description can be found at:

```
http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html
```

The Java policy tool (`policytool`) can be used to easily create a policy file.

To provide a policy file to a Java class, two methods can be used:

- Set the property java.security.policy at the command line, like this:
  `java -Djava.security.policy=<policy file>`

- Explicitly set the property from within the Java class, like this:
  `System.setProperty( "java.security.policy", "<policy file>" );`

The server side execution requires an additional property to be set: the codebase annotation. The codebase annotation is the url that is stuck to every instance of MarshalledObject that is created within a JVM. This url must be a meaningful url from the client's point of view, so the use of schemes like file:// is forbidden unless it is known for sure that the client will find the classes at the specified path of its own file system. The preferred way to publish class files is through an HTTP server, although the same restrictions about public availability apply.

For instance, if the server is in a network that provides an HTTP server that is accessible to its clients as `http://mynet.mydomain.com`, and the virtual HTTP server path `/classes` points to a directory where the jar file containing the classes for the server proxy can be found, a valid codebase annotation would be `http://mynet.mydomain.com/classes/serviceProxy.jar`. Be specially careful with urls that contain localhost, 127.0.0.1 or otherwise any private IP numbers. If they are used as a codebase annotation the MarshalledObjects won't be usable outside the scope where these urls are valid.

To set the code base annotation the same mechanisms as for the policy file apply. That is, there are mainly two ways to set the codebase annotation:

- Set the property java.security.policy at the command line, like this:
  `java -Djava.rmi.server.codebase=<url annotation>`

- Explicitly set the property from within the Java class, like this:

```
System.setProperty(
  "java.rmi.server.codebase",
  "<url annotation>"
  );
```

***A word of advice**: the setting of any Java property must be done BEFORE setting the security manager. The security manager forbids*

the setting of any Java property unless explicitly allowed by the policy file.

As a deployment and execution example, the server class will run on a host called `myhost.mynetwork.com`, in port 9999. The service proxy classes will be stored in a jar file called `printer-dl.jar`, which will be accessible by http in the directory classes of the http server at `http://mywebserver.anothernetwork.com`, in port 8000. The location of the client is indifferent, as long as it can access the aforementioned web server and the FADA network.

The invocation of the server class, provided the code above is used, is:

```
java -Djava.security.policy=./policy.file
-Djava.rmi.server.codebase=
http://mywebserver.anothernetwork.com:8000/classes/printer-
dl.jar
-classpath .:fada-toolkit.jar Printer 9999 myhost.mynetwork.com
fada.fadanet.org:2002
```

The Java properties are set in the command line, and therefore take effect before the security manager is instantiated and set. The policy file property points to a file in the local directory that is called policy.file, and contains the policies needed to execute code related to the FADA. A description of such file is given in the chapter [FADA security wrappers].

Then comes the codebase annotation. The codebase annotation, as mentioned above, is:

`http://mywebserver.anothernetwork.com:8000/classes/printer-dl.jar`.

Note that it does not end in a slash, as the codebase is referring to an actual file (the file `printer-dl.jar`) and not a directory.

The next parameter is the classpath. Note that it needs both the actual directory, where the classes Printer and PrinterProxy are, as well as the `fada-toolkit.jar` file, where the FadaHelper, FadaLeaseRenewer and related classes and interfaces are.

Then the class name is given for invocation. The rest of the line are the parameters to the server class. In this example, first comes the

port number (9999), then the host where the server is (`myhost.mynetwork.com`), and then the url of a known FADA node.

Note that the host name is `myhost.mynetwork.com`, and that things such as localhost, 127.0.0.1, 192.168.0.1, which are private network designations, and the like won't work unless the client is also in the same host as the server, which makes the whole business of creating a distributed application pretty pointless.

The client can be invoked by using the following command line:

```
java -Djava.security.policy=./policy.file
-classpath .:fada-toolkit.jar PrinterClient myfile.txt
fada.fadanet.org:2002
```

First note that the client doesn't express a codebase annotation. The client requires none, because in this example the client is not exporting any class. However, the policy file is needed because the client is *USING* downloadable code.

Then the classpath is specified. As above, the actual directory is used, provided that the ProxyClient class resides there. The RemotePrinter interface is also needed in the classpath, because that class is not downloaded, but rather used directly in the client code. This is common to all FADA applications: the interface is used directly in code and therefore is needed for both compilation and execution.

After the name of the class to invoke there are two parameters. The first one is the name of the file to print. In this simple example it has been assumed that the remote server is able to print only text files, and so only text files must be sent by using this client application.

The second parameter is the url of a known FADA node. Note that it is not the same as the url for the FADA node provided in the server side. In spite of that, the client will find the proxy for the desired service if there exists a path between the FADA node whose url is `fada.fadanet.org:2002` and the FADA node whose url is fada.fadanet.org. Don't be misled by the fact that both FADA nodes belong to the same domain. The existence of a path between two given FADA nodes depends on the existence of neighbor FADA nodes that are connected to both of them, and there is a path through the neighborhood of FADA nodes between the two.

In this simple example the remote method does not return anything, so there is no way to get confirmation or denial from the server side. However, this is of little interest as this example relates to the FADA architecture and its software toolkit, and therefore has intentionally been left out of this document.

The FADA stub - skeleton compiler: an example

Easier development of the server: use of fadagen

The previous section dealt with the generic use of the FADA to provide a client-server application to be deployed using the FADA. The service proxy was explicitly created, and thus it was shown the flexibility achievable with the use of proxies. The service proxy had plenty of flexibility about what protocol to use to communicate with the server side, and the server side had plenty of flexibility about how to implement the service. In fact, in the example the details of how to print the file were left out intentionally. One possible solution would be to use the Java Native Interface (JNI) to make the server class, written in Java, to interact with some function written in C or other language and stored in a library file (a DLL file in Windows platforms, a .so file in UNIX environments). Another possible solution would have been to directly interact with the print server from within the Java application, by sending the bytes to a defined port, or writing the file to a spooling directory, etc. All these details were hidden from the client's point of view.

A common way to develop distributed applications in Java is through the use of RMI. But it has already been seen the problems it poses. The FADA development team provided a partial replacement of the RMI framework, and in this section such replacement will be further explored.

The previous example will be modified to allow the use of the FADA stub/skeleton compiler (fadagen). This modification requires changes in the server side class, and also in the service proxy class, but NOT on the client side. The abstraction layer provided by the use of a service proxy effectively isolates the client from the server side implementation details. That's exactly what allows clients of Jini in general, and FADA in particular, to use the same client code for different server implementations.

As the section about the FADA stub/skeleton compiler explains, the server class must implement the `net.fada.remote.Remote` interface, and all remote methods must be accordingly tagged by stating that they throw the `net.fada.remote.RemoteException` exception in their throws clause. A new interface will be declared.

The new interface is:

```
import java.io.*;
import net.fada.remote.*;

public interface RemotePrinterWithStub extends
net.fada.remote.Remote {
  public void print( byte[] contents )
  throws IOException, net.fada.remote.RemoteException;
}
```

The interface now extends the interface net.fada.remote.Remote. By making this change it will be possible to run the fadagen utility on the server side class, when it has been changed. The print method now throws `net.fada.remote.RemoteException` as well as `java.io.IOException`. This is to make the fadagen utility aware that code to make this method remote must be provided.

```
import java.net.*;
import java.io.*;
import net.fada.toolkit.*;
import net.fada.directory.tool.*;
import net.fada.remote.*;
import net.fada.transport.*;

public class Printer
extends RemoteObject
implements RemotePrinterWithStub {

  int portNumber;
  ServerSocket serverSocket;
  Thread runner;

  public Printer( int portNumber ) throws IOException {
    this.portNumber = portNumber;
    this.serverSocket = new ServerSocket( portNumber );
  }

  public void print( byte[] contents )
  throws IOException, RemoteException {
```

```
    // Implementation details of the print method
    // have been intentionally left out of the example

  }

  public static void main( String[] args ) throws Exception

    // Get parameters
    // First parameter is the port number

    int portNumber = Integer.parseInt( args[0] );

    // Second parameter is the server host name, needed for the
proxy

    String url = args[1];

    // Third parameter is the url of the FADA node
    // Discovery mechanisms could be used here

    String fadaUrl = args[2];

    // Create the instance of the server

    Printer me = new Printer( portNumber );

    // Prepare to create instance of the stub

    ServerTransport st = new ServerTransportImpl( portNumber );
    ClientTransport ct = new ClientTransportImpl( url );

    // Create the stub instance

    RemotePrinter stub = (RemotePrinter) me.export( st, ct );

    // Register the stub in the FADA

    // Prepare lease renewer

    FadaLeaseRenewer flr = new FadaLeaseRenewer();
```

```
    // Prepare the FadaHelper instance

    FadaHelper fh = new FadaHelper( flr );

    // Register the service proxy in a FADA node

    FadaServiceID id = null;
    FadaLookupLocator locator = new FadaLookupLocator(fadaURL);
    id = fh.register(
      locator.getRegistrar(),
        (Serializable)stub,
        null,
        new String[] {PrinterProxyOnlyStub},
        10000L,
        null,
        “http://server:port/where_codebase_is.jar”,
        null );
  }

}
```

Note that, as in the previous example, the implementation details about the actual printing of the file are not specified.

The server class must extend the class `net.fada.remote.RemoteObject`, because this class offers the export method. This method takes an instance of two classes, which are implementations of the interfaces ServerTransport and ClientTransport. The FADA software bundle offers a default implementation for each interface, using HTTP as the transport layer. The implementation of the transport layer is open, and can be freely modified by providing a different implementation of the ServerTransport and ClientTransport interfaces.

The other reason why the server must extend a predefined class is that the class `net.fada.remote.RemoteObject` has the needed functionality to provide a concurrent server of remote requests. Note how all the code involved in the control of separate requests has been eliminated from the server class, leaving a much clearer implementation, only concerned with the business logic.

It must be noted that this is a long-lived service, and therefore it doesn’t deregister itself from the FADA, but rather continues execution undefinitely. A real service should offer some mechanism

for the administrator to take it down (for administration tasks, migration to another platform, whatever).

The client code is exactly the same, save by two details:

- The client code uses the FadaHelper to look in the FADA for a different interface, this time being RemotePrinterWithStub.

- This interface has a different signature for the method print, and therefore the client is required to provide proper parameters, otherwise it can't be called.

This approach has definitely exposed some implementation details to the client, hardly a desirable situation. An alternative and better approach would be to use a mixture of the approaches above. That is, to provide a service proxy that uses the stub obtained on the exportation of the `net.fada.remote.RemoteObject`. In this way, if an implementation detail changes, it is only needed to change the code for the service proxy, and not the interface and the client as well. Also, by keeping the stub and skeleton provided by the fadagen application, it is possible to program servers in an easy way, without the need to deal with concurrent server issues.

As the skeleton and stub use by default a transport layer that's based on HTTP, it is easy to deploy such an application with clients behind firewalls, because if on such environment there is a gateway or proxy server that allows the crossing of HTTP messages across the firewall boundaries, it is possible to provide the complex functionality of a full-fledged server without the need to open extra ports on the firewall, which makes the administration of the networking environment easier.

Skeleton/stub and proxies

In the previous example the service interface had to be changed to allow the use of the fadagen utility. This forced the change of the client code, an undesirable side effect. In this section another approach will be used, taking the best of both worlds: easy server class with the use of fadagen, and an unaware client with the use of the service proxy.

The service interface defines what the client may or may not know about the service. It is the only piece of code the client has actually to know in order to use the service proxy. Unfortunately, service interfaces for use in FADA are not aware of the existence of fadagen (the fadagen utility saw the light long after the FADA was used in

testing environments). Therefore these service interfaces are not well suited for use with the fadagen utility.

But there is one thing that can be done to use both the service proxy that implements the service interface and to use the fadagen utility to easier develop the server side class: to write a service proxy that uses the stub generated by the fadagen utility.

The role of the service proxy is to provide a bridge between the service interface (known to the client) and the server implementation. Therefore the natural way to isolate the changes caused by the use of the fadagen utility, is to write a service proxy that implements the service interface, therefore effectively isolating the client from server implementation details.

The new service proxy would be like this:

```
import java.io.*;
import java.net.*;

public class PrinterProxy implements RemotePrinter, Serializable
{

  RemotePrinterWithStub stub;

  public PrinterProxy() {
    // For serialization purposes
  }

  public PrinterProxy( RemotePrinterWithStub stub ) {
    this.stub = stub;
  }

  public void print( File file ) throws IOException {

    // Prepare to read the file
    FileInputStream fis = new FileInputStream( file );

    // Read the file
    byte[] chunk = new byte[ 1024 ];
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    int read = -1;
    while( ( read = fis.read( chunk ) ) != -1 ) {
```

```
        baos.write( chunk, 0, read );
      }

      // Call the server
      try {
        stub.print( baos.toByteArray() );
      } catch( Exception ex ) {
        throw (IOException) new IOException(
          ex.getMessage()
        ).fillInStackTrace();
      }

      fis.close();
      baos.close();
      s.close();
    }

}
```

The print method is now compliant with the service interface, and therefore the change in the server is invisible to the service client. The operation of reading the file is now performed by the service proxy, and not by the client as was the result of using the server stub. That is exactly the role of the service proxy: to perform the small set of operations that may be needed in order to adequate parameters known to the client to parameters usable by the server side, and to transform result from the server side to results suitable to the client side. In this case the server returns no results, and therefore the service proxy hasn't to do anything about it.

The service proxy also isolates the client from the exceptions thrown by the server stub that are not declared in the service interface. If such an exception is thrown by the stub, it is caught by the service proxy and wrapped within an exception the client is aware of. The client is protected from implementation-specific exceptions.

The service proxy must be created and registered by the server side, and therefore the server code must be changed yet once again. Here is how it looks:

```
import java.net.*;
import java.io.*;
```

```
import net.fada.toolkit.*;
import net.fada.directory.tool.*;
import net.fada.remote.*;
import net.fada.transport.*;

public class Printer
extends RemoteObject
implements RemotePrinterWithStub {

  int portNumber;
  ServerSocket serverSocket;
  Thread runner;

  public Printer( int portNumber ) throws IOException {
    this.portNumber = portNumber;
    this.serverSocket = new ServerSocket( portNumber );
  }

  public void print( byte[] contents )
  throws IOException, RemoteException {

    // Implementation details of the print method
    // have been intentionally left out of the example

  }

  public static void main( String[] args ) throws Exception {


    // Get parameters
    // First parameter is the port number

    int portNumber = Integer.parseInt( args[0] );

    // Second parameter is the server host name, needed for the
proxy

    String url = args[1];

    // Third parameter is the url of the FADA node
    // Discovery mechanisms could be used here
```

```
String fadaUrl = args[2];

// Create the instance of the server

Printer me = new Printer( portNumber );

// Prepare to create instance of the stub

ServerTransport st = new ServerTransportImpl( portNumber );
ClientTransport ct = new ClientTransportImpl( url );

// Create the stub instance

RemotePrinterWithStub stub =
  (RemotePrinterWithStub) me.export( st, ct );

// Create the service proxy

RemotePrinter proxy = new PrinterProxy( stub );

// Register the service proxy in the FADA

// Prepare lease renewer

FadaLeaseRenewer flr = new FadaLeaseRenewer();

// Prepare the FadaHelper instance

FadaHelper fh = new FadaHelper( flr );

// Register the service proxy in a FADA node

FadaServiceID id = null;
FadaLookupLocator locator = new FadaLookupLocator(fadaUrl);
id = fh.register(
     locator.getRegistrar(),
     proxy,
     null,
     new String[] {PrinterProxySmartStub},
     10000L,
```

```
            null,
            “http://server:port/where_codebase_is.jar”,
            null );
    }

}
```

Notice how the interfaces implemented by the server class and the service proxy are not the same. This is quite straightforward, since the client should know nothing about the implementation details of the server class, and therefore it is irrelevant if the server side class implements or not the service interface. In fact, the server side needn’t be a Java class, as long as there is a way to register a service proxy from without a Java class, and, more important yet, to continually renew the lease from within a non-Java executable.

The use of JNI could make this possible, but that is left to the developers of the server side software if they wish so. The use of JNI requires a per-platform development and compilation of bridges between the native platform in use and the FADA class methods, plus the creation of shared libraries for each platform, and therefore it is not possible to provide an API with JNI methods for all platforms. Moreover, the execution of such shared library objects is platform dependent, and therfore it is not possible to provide a general implementation. Server side developers may develop a JNI wrapper for the FADA toolkit classes and methods, in the best way that is suits their platform environment.

As for the use of JNI for the client side, it is not possible to provide a service proxy that uses JNI (or that can be used through JNI) for all possible client platforms. This is exactly the reverse scenario from the paragraph above, and it’s much more complex to deal with. The server side has a fixed platform, so the use of JNI could be theoretically used. But when it comes to the client side, it is the service provider that must develop the service proxy. If the service proxy uses the JNI for the client environment the service proxy must be able to cope with all client platforms, a situation that it’s not possible to cope.

The client side code for the latest example is the same as in the first case, because that is exactly the scenario: a service proxy that implements the service interface. The example where the client uses directly the stub is not recommended unless the service interface can be readily used as an interface for the fadagen utility.

FADA security wrappers

Distributed computing is fundamentally different from centralised computing. The usually mentioned four major differences include latency, memory access, partial failures, and concurrency. Security should definitely be added to this list, since a distributed system requires cryptography to be used while a centralised system may survive without it.

Since the creation of computer networks, security has been an important concept within these systems. The need for having different and disparate systems interconnected forces us to put a great effort to protect the use of shared distributed resources of malicious intentions.

In reality, computer networks are insecure, and some security features are desired. For example, the service may wish to authenticate clients, and based on who the client is, allow some operations and deny others. In distributed systems, this functionality is achieved using cryptographic protocols. For example, the Transport Layer Security (TLS) protocol supports authentication of both the client and the server using public keys and X.509 certificates.

The security mechanism in FADA is strongly associated with the use of certificates. In fact, strongly associated with the use of X509 certificates, which are mainly supported by the JAVA standard API and used worldwide. The use of this type of certificate is due to the use of a PKI cryptographic system in the signatures of all objects that are moving inside the FADA network.

In this paper, the concept of FADA federations is explained around FADA security. We will see the meaning of the phrase “We only trust in those entities whose are certified by the same entities in which we trust”.

Security background

In this chapter, firstly, it is given an overview of relevant concepts and technologies from the field of computer security. The rest of the chapter describes existing Java technologies, including the Java 2 security architecture.

Computer Security

Computer security deals with the prevention and detection of unauthorized actions by users of a computer system. Usually, this includes at least the protection of confidentiality, integrity, and availability. Sometimes accountability, and even dependability, are also added to this list. In actual systems, the protection of these properties are achieved through various security services and mechanisms. From the point of view of this document, the most important services are authentication, authorization, and access control. In this work, authentication means verifying a claimed identity. Authorization means granting access to a restricted resource to someone, and access control mechanisms enforce these restrictions. In distributed systems, these functions are usually supported by various cryptographic primitives and protocols.

Decentralized trust management

Traditionally, access control has been based on identity authentication and locally stored access control lists (ACLs). The most popular method for identity authentication is probably user names and passwords. Another widely used method is to rely on public keys with identity certificates. Basically, identity certificates, such as X.509, bind a human-readable name to a public key. It is important to notice that these certificates are fundamentally different from authorization certificates, described below.

Access control lists describe what access rights a user has for a resource. For instance, an entry in a list can grant Alice a read permission to some file. However, when applied to a distributed system, the ACL approach has a number of drawbacks. For instance, operations which modify the access control list need to be protected

somehow. To illustrate this issue, the following example is given by Ellison et al.

"... Imagine a firewall proxy permitting telnet and ftp access from the Internet into a network of US DoD machines. Because of the sensitivity of that destination network, strong access control would be desired. One could use public key authentication and public key certificates to establish who the individual keyholder was. Both the private key and the keyholder s certificates could be kept on a Fortezza card. That card holds X.509v1 certificates, so all that can be established is the name of the keyholder. It is then the job of the firewall to keep an ACL, listing named keyholders and the forms of access they are each permitted.

Consider the ACL itself. Not only would it be potentially huge, demanding far more storage than the firewall would otherwise require, but it would also need its own ACL. One could not, for example, have someone in the Army have the power to decide whether someone in the Navy got access. In fact, the ACL would probably need not one level of its own ACL, but a nested set of ACLs, eventually reflecting the organization structure of the entire Defense Department."

Indeed, Blaze et al. argue that "*the use of identity-based public-key systems in conjunction with ACLs are inadequate solutions to distributed (and programmable) system security problems*".

Trust management, introduced by Blaze et al., proposes an alternative solution. Basically, trust management uses a set of unified mechanisms for specifying both security policies and security credentials. The credentials are signed statements (certificates) about what principals (users) are allowed to do. Thus, even though they are commonly called certificates, they are fundamentally different from traditional name certificates. Usually the access rights are granted directly to the public keys of users, and therefore trust management systems are sometimes called key-oriented PKIs.

The unified mechanisms are also designed to separate the mechanism from the policy. Thus, the same mechanisms for verifying credentials and policies (a trust management engine) can be used by many different applications. This is unlike access control lists whose structure usually reflects the needs of one particular application.

Java Security Architecture

The FADA architecture relies heavily on the security features in the base Java technology. Security was one of the main goals in the design of the Java language and execution environment. The security features were originally designed for applets that is, small applications embedded inside web pages but have since then received numerous other uses. When running inside a web browser, an applet should not be allowed to access sensitive resources, such as the user s files, or open arbitrary network connections, for instance. The security architecture of Java 2 consists of the following components.

- Java language and platform: type safety and isolation.
- Resource access control: policy and enforcement.
- Cryptography architecture.

The Java language and platform security are described in the next section, and resource access control in later section. The third important component is the cryptography architecture. It provides access to cryptographic algorithms, such as message digests, digital signatures, symmetric and asymmetric ciphers and key agreement algorithms. It is used as a building block in the construction of the other security mechanisms.

Java language and platform security

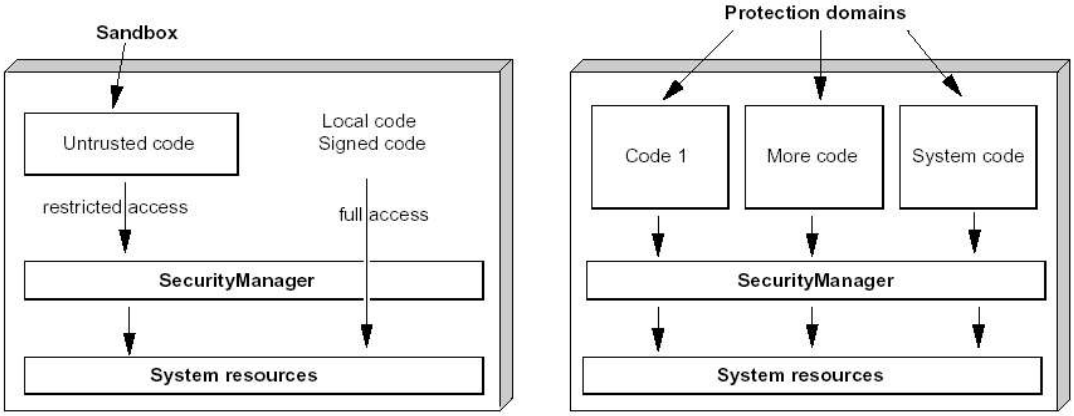
The Java language is designed to be type safe. This means, for instance, that no Java program can ever refer to an object using an incorrect type, refer to an unassigned memory location, or forge pointers from integer types. Also, access restrictions (private, public, package local) on classes, methods, and fields cannot be violated. Some of these type checks are performed by the compiler, but Java is usually compiled into an intermediate platform-neutral form called byte code. This intermediate form is interpreted by a Java Virtual Machine (JVM). The actual checks must be performed on the byte code, since it is possible to bypass the compiler and write byte code by hand. In the JVM, type safety is implemented using runtime checks (for example, type casts) and the byte code verifier. The byte code verifier checks the code when it is loaded, and ensures that it respects the Java language rules. The byte code verifier is a very complex piece of code, and most of the security bugs found in Java

implementations so far have been in the byte code verifier. In addition to type safety, untrusted code needs to be isolated. In Java, the isolation is provided by class loaders. Class loaders are responsible for mapping class names (e.g., `java.lang.String`) to the corresponding byte code, and loading the byte code from a file or from the network. The mapping is context-dependent: there can be two classes with the same name running inside a single JVM, provided that they are loaded with different class loaders. The class loaders are themselves written in Java, and programmers can write new class loaders, if necessary. Class loaders also interact with type safety. Because there can be more than one class with the same name, references to names must be resolved consistently, i.e., in a way which preserves type safety.

Resource access control

The resource access control framework is responsible for controlling access to valuable system resources, such as the file system. This part of the infrastructure has considerably evolved during the history of Java: both the enforcement and policy mechanisms are now more flexible and fine-grained than in the original Java 1.0. In JDK 1.0 and 1.1, all code was either untrusted or completely trusted. Untrusted code was run side a sandbox, which limited its access to sensitive operations. In JDK 1.0, all code loaded from the local file system was considered trusted, and everything else (e.g. loaded from the network) untrusted. However, sometimes applets have a legitimate need to access some protected resources. Thus, JDK 1.1 introduced the notion signed applets. In Java the byte code for an application is usually stored in a Java archive (JAR) file. The JAR file can also include a digital signature. If the JAR file was signed by a trusted key, the code was considered trusted, even if the JAR file itself was loaded from the network.

JDK 1.1 model (sandbox) and JDK 1.2 protection domains compared. In JDK 1.1 the code is either in the sandbox or trusted. In Java 2 the code is divided into different protection domains.



JDK 1.1 JVM JDK 1.2 (Java 2) JVM

In Java 2 (JDK 1.2 or greater) the security architecture was almost completely redesigned. Code is no longer treated simply untrusted or completely trusted, but is divided into protection domains. All the code running inside one protection domain share the same access permissions, but there can be as many protection domains as necessary. Classes are assigned to protection domains based on the URL and digital signatures of the JAR file. The permissions granted to protection domains are also more fine-grained than in JDK 1.1. For instance, it is possible to grant a permission to read only a particular file, or open a network connection to a single port on a single host. The set of permissions is not fixed but can be extended by programmers to protect application-specific resources. When checking permissions, the access control mechanisms check the Java call stack. The effective permissions are the intersection of the permission of all the classes in the call stack, from the topmost to the first privileged stack frame, or the whole stack if none of the frames is marked as privileged. Privileged frames allow a piece of code to perform some operation with its own privileges, regardless of who originally called it.

Requirements for FADA Security

Before designing a security framework for FADA, it is necessary to decide what kind of security functionality is required. Naturally, this depends on the concrete applications written using FADA and on the trust relationships involved. We can differentiate two scenarios in the FADA architecture:

- Interaction between FADA nodes.
- Interaction between users of FADA network (Service providers and service users) with FADA nodes.

Security frameworks of both scenarios are different because the security functionality in each scenario is not the same.

In the real use of applications inside the FADA network there is another scenario, the interaction between service users with service providers, but this scenario is not inside the scope of security framework of FADA, but is a task of the service provider to give this framework for a correct and secure use of their services.

FADA network security functionality

The requirements for FADA network security functionality are the following:

- A FADA node must be able to avoid interaction with other FADA nodes that aren't nodes of its federation. These interactions may be a service lookup extension, a request for connection, and so on.
- A FADA node is not the responsible of discarding malicious service proxies stored within them. The service user is the last chain link when it is made a lookup for a service proxy. It must have the option of discarding the service proxies in which it does not trust.
- A FADA node can belong to different federations, as long as these federations are in the same hierarchy of federations. In terms of security, a FADA node can belong to all federations which trust in its

certificate. In the next chapter the concept of federations is explained more in detail.

- The FADA network is an heterogeneous entity, so each node present in the FADA network takes care of its own security, without assuming any rule applied with itself or with other nodes.

- Each FADA node is free to be implemented within its own mechanism of security, which is represented in its FadaProxy. The only restriction is that the FadaProxy methods must fulfill the semantic of the FadaInterface.

- A Fada node can deny the use of its functionality to non-trusted users. The verification and authentication of these users may be internally managed by the FadaProxy and must be transparent to the user.

FADA's users security functionality

The users of FADA, like service providers and service users, interact with the FADA network to register services or to retrieve services to be used by them. To achieve this, they must fulfill some requirements to guarantee the correct accomplishment of FADA security rules.

The requirements are explained in the following lines:

- The service providers can only register their service proxies in any FADA node federation that trust in its certificate. Otherwise, the registration may not be completed successfully because this action is rejected by FADA node side.

- The service provider may discard untrusted Fada proxies when registration is attempted.

- The service user may discard untrusted service proxies obtained during a lookup action. This entity is the last link chain when it is made a lookup for a service proxy, and so, this entity must have the possibility of discarding service proxies that are not signed by the certificate in which this entity trusts.

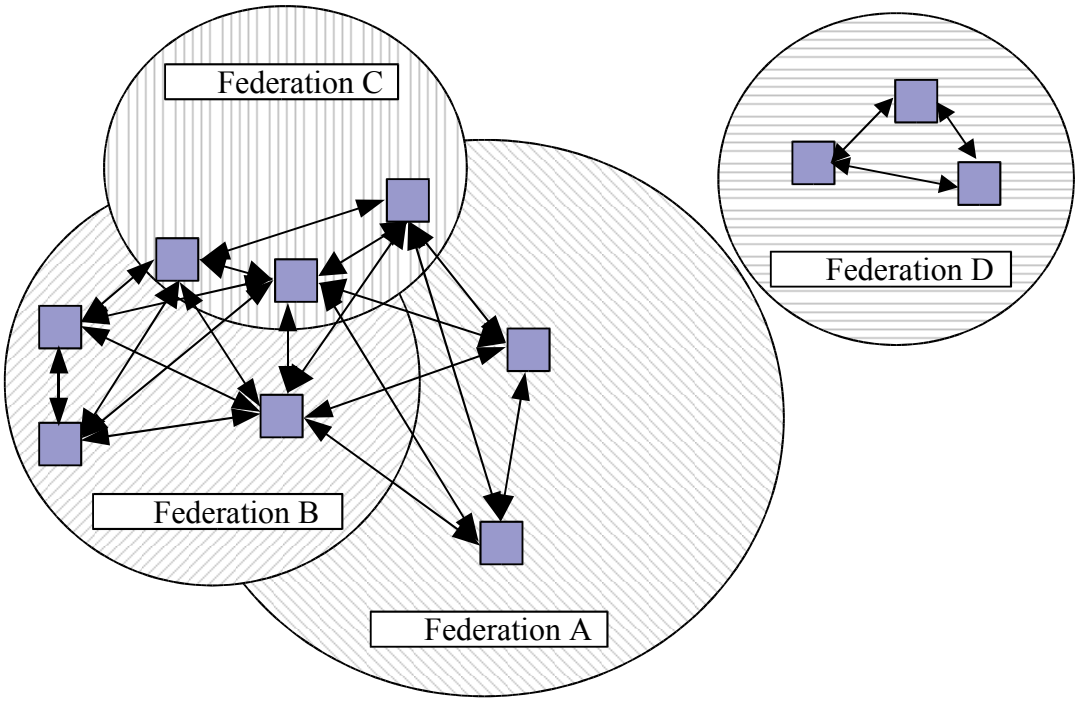
FADA and Federations

The concept of federation is very simple: several entities that work together to perform a particular functionality. It is possible to extend the meaning of this concept, if the particular functionality must be made by a selected group of entities that have a particular relationship.

A FADA federation is a group of FADA nodes that trust one another. This trust can be described as a trust relationship.

Trust relationships are based in the trust of certificates. So, the federations are created according to the existence of hierarchies of certificates. This hierarchies can be seen as certificate chains which go from the root certifier authority (last certifier entity, which certifies other entities using a self-signed certificate) to target entity (first entity in the chain).

The following graphic illustrates the existence of different federations of FADA nodes and which are the trust relationships among them.



Trust relationships diagram

In this diagram we can see a sample scenario with different federations. The rules for a federation's creation is related with the use of certificates signed by certifier authorities (CA's) by the FADA nodes.

For each certifier entity that certifies a FADA node (or some of them) a federation is potentially created. This certifier entities may be root certifier authorities (which have a self-signed certificate for certifying) or intermediate certifier authorities (which have a certificate signed by another certifier authority for certifying).

In the above diagram the Federation A is formed by FADA nodes that use certificates signed by a root CA. On the other hand, the Federation B is formed by FADA nodes that uses certificates signed by an intermediate CA (this CA is certified by the root CA cited previously).

A FADA node can belong to different federations according to the certificates in which it trusts. In the diagram the FADA nodes that are into a circle intersection belong to different federations. This means these FADA nodes trust in the certificates associated to both federations.

Trust relationships between FADA nodes exist if the followings cases are given:

- A FADA node in federation X trusts all FADA nodes belonging to this federation X.
- A FADA node in federation X may trust FADA nodes belonging to another federation Y if and only if FADA nodes in federation Y trust the certificate associated to federation X.

Dynamic and secure loading of remote code in FADA

When the security wrapper was planned for the FADA, different mechanisms were proposed to provide a trustworthy security that were as much transparent to the user as possible.

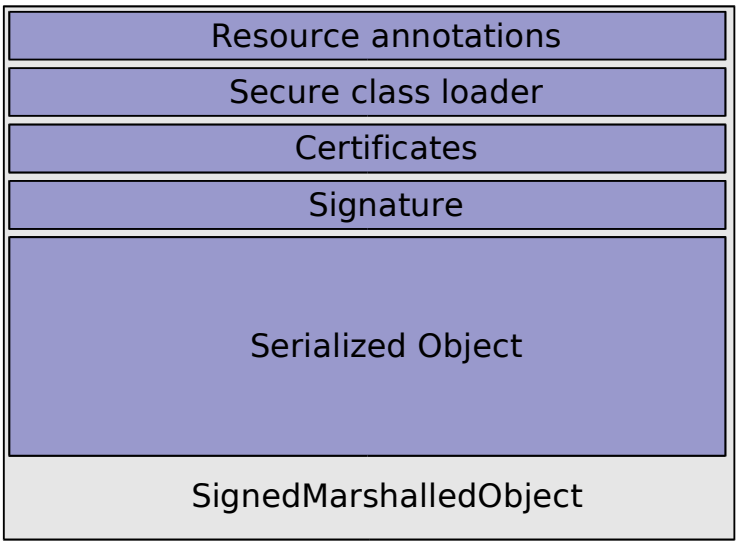
The more crucial part, in terms of security, in a distributed system is, without a doubt, the execution of remote code in the local JVM. For this, the security mechanism has been designed around this point. The security mechanism is totally integrated in the side which is in charge of download and execution of the remote code. The low level nature of this mechanism allows high security and remains transparent to the user.

SignedMarshalledObject

While RMI provides the java.rmi.MarshalledObject class that makes possible all mechanisms of downloading remote code and its following load and instantiation of this remote code in the local JVM, for communications in FADA a new class `net.fada.directory.SignedMarshalledObject` has been created. This class inherits the concepts and mechanisms used in the RMI class, but new functionality has been added that allows to outfit a secure nature to the load and execution of remote code in the local machine.

SignedMarshalledObject's structure

The SignedMarshalledObject's structure is defined in the next graphic:



SignedMarshalledObject object structure

Like the annotation field in the `java.rmi.MarshalledObject`, the annotation field in the SignedMarshalledObject also specifies the possible locations of resources (classes) that allows the correct deserialization of the object which is serialized inside of SignedMarshalledObject. However, in the case of the SignedMarshalledObject, only jar files can be the annotation, because they can be signed, and obviously, directories can not.

The class loader is the main part of this object, which is in charge of loading the marshaled object into the local JVM. This customized class loader is responsible for checking all security rules which have to apply before the load of the remote object into the JVM.

The signature is the encrypted digest made of the bytes of the serialized object. Within the verification of this signature it is possible to ensure the authenticity and integrity of bytes of the object serialized into the SignedMarshalledObject. This field is an instance of the class `java.security.Signature`, present in the standard JAVA API. Besides, this object contains a byte array (which gives the name of signature), and it is also in charge of verifying the cited signature.

The certificates field is the certificate chain that authenticates this object and allows the verification of the previous signature.

Finally, the last field is the serialized object, as an array of bytes.

There are parts of the SignedMarshalledObject that have been intentionality omitted in its structure description because their functionality is very specific, like the object's deserialization or caching of jar files, which have not direct impact in the application of security rules.

Step by step: Secure instantiation of a remote object in FADA

The next paragraphs explain in detail how the instantiation of a remote object is made in FADA, giving more details in implicated parts that have reference to the security.

The steps to follow in the instantiation of a remote object in FADA are:

1 Unmarshaling the object.

1.1 Signature (of the serialized object's bytes) verification.

1.2 Deserialization and load of the object

1.2.1 Downloading of needed classes for the instantiation of the serialized object.

1.2.2 Downloaded code verification.

1 ***Unmarshaling the object***.

The marshaled object may be obtained in differents ways, according to the scenario in which we are involved, specifying more, according to the object that would like to be obtained. There are two types of objects which are marshaled in FADA. One of these is the object that takes over the communication with a FADA node (FadaProxy), and the other type are the service proxies. To retrieve a FadaProxy an HTTP request is made through TCP connection to the node which

we'd like to communicate with. This FADA node returns in the same communication channel its FadaProxy marshalled as array of bytes. On the other hand, the achievement of a service proxy is always made by a FadaProxy, and so, the way to obtain this service proxy depends on the implementation of the FadaProxy we are using to make this action.

The methods for unmarshaling an object are the following:

- `public Object SignedMarshalledObject.get();`
- `public Object SignedMarshalledObject.get( Certificate certificate );`

Both return an object java.lang.Object. This makes necessary a later casting to the correct object type. As we can see, the `get()` method is overloaded, because there is the possibility of unmarshaling the object applying or not the security rules. If a certificate is passed in to the method, the security rules will be applied. But if no certificate is passed in or a null object is passed in to the method the security rules will not be applied.

From now on, it is supposed that a valid certificate has been passed in to the method get to unmarshal the object.

1.1 ***Signature (of the serialized object's bytes) verification***.

When the marshaled object has been obtained, the first thing that must be done is to check if the certificate passed is contained in the certificate chain embedded in the marshalled object, and if the certificate chain is well-formed. If any of these verifications fail, the unmarshalling will be stopped and a exception will be thrown signaling that it is doesn't trust the marshalled object.

In this point the integrity of contents of the serialized must be checked. To do that, the signature embedded in the SignedMarshalledObject is verified. This signature must have been created from the bytes of the serialized object. To verify this signature, it must be passed in to the signature verifier (the signature object itself) the public key contained in the first certificate of the certificate chain embedded in the marshaled object. If the verification fails, it is no able to authenticate the serialized object, and so, the deserialitation will be stopped and an exception will be thrown signaling this failure.

1.2 ***Deserialization and load of the object***

If the integrity and authentication of bytes of the serialized object has been verified, the next step is the deserialization of this object. During this deserialization it will be necessary the definition and load of some classes needed for cited deserialitation. As a minimum, the class of the serialized object will be defined and load, as well as some classes which are referenced in this class. Here the customized class loader enters the picture. It attempts to find the class bytes to define all classes needed to complete the deserialitation of the serialized object, as the class this object belongs to and all the related classes may have not been loaded to this JVM yet.

1.2.1 Downloading needed classes for the instantiation of the serialized object

When the class of the serialized object, as well as the classes referenced by this class, are not found in local resources (classpath), the customized class loader must take the annotations present in the marshaled object to know where it must search the remote code that defines the cited classes.

These annotations must be references to accessible URLs so the customized class loader may download the files where the code that defines the classes needed for deserializing the object embedded in the marshaled object can be found. If any of the needed classes is not found in any resource referenced in the annotations, the load will stop the class loading of the serialized object and an exception will be thrown signaling that a needed class for the definition of the serialized object can not be found.

1.2.2 ***Downloaded code verification***.

To verify the downloaded code by the customized class loader it is necessary that each class has an associated signature that signs it. So it is possible to check, as it has been done with bytes of the serialized object, its authenticity and its integrity. For this reason it has been taken the design decision that the classes downloaded must be stored in a jar file (java archive file). This jar file must be signed with the same private key used to sign the bytes of the serialized object. So, it is possible to authenticate that the source of the instance and the code is the same entity, without any kind of doubt.

For more details about signing jar files see [44].

If the jar file is corrupted or any signature inside the jar file can not be verified the deserialization and the loading will be stopped and an exception will be thrown signaling that trusted code for deserialization of the marshalled object can not be found.

If all processes have been successfully completed, the object has been finally unmarshaled and is returned by the method. It must have in account that all marshalled objects should be implementations of defined interfaces. These interfaces must be present in the entity that would like to unmarshall implementations of them, but the following casting can not be made with the returned object until after unmarshalling the object.

A main goal, for the entities who would like that others use their services proxies, is to do a good job where the interface of the proxy has to be defined. This definition must be well written enough to avoid later changes in case of adding new functionality to the proxy (the class which implements the interface). If a new functionality has to be added to the proxy but no changes have to be made to the interface it means that a great job has been made in the definition of the interface. We must remember that the implementation of the interface is dynamically loaded every time the proxy has to be used while the interface for this proxy is loaded one time during the life time of the entity who needs to use the cited proxy.

SignedMarshalledObject instantiation

For someone to be able to use the services of other entities, these entities must first create their proxies which will be acting as a gateway between the provider entity (entity who provides the service) and the target entity (entity who uses the service). But all objects which move within the FADA network must be marshaled, these service proxies should be marshaled before they can take part inside the FADA network, and so, to be used by other entities interested in these services.

For marshaling an object into a SignedMarshalledObject there are two possibilities:

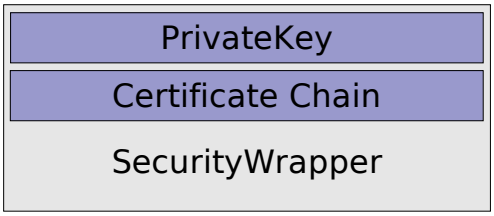
- ```
  SignedMarshalledObject smo =
  new SignedMarshalledObject( Object objToMarshall );
  ```
  ```

With this first option no signing will be applied to the object. For this reason, it will be only used by the entities which do not require the use of secure proxies, it means, proxies to which the security rules can not be applied.

```
SignedMarshalledObject smo =
  new SignedMarshalledObject(
    Object objToMarshall,
    SecurityWrapper wrapper );
```

With this second option it will be possible to be potentially used by all entities (entities which require secure proxies and entities which don't have this security requirement).

Here a new kind of object enters the scene: the SecurityWrapper object located in `net.fada.directory.security` package. This kind of object is an object which is used as a container for two objects needed for signing and verifying the proxy (exactly, bytes of the serialized proxy).



SecurityWrapper object structure

This object can be created through the following constructors.

- `public SecurityWrapper ( PrivateKey key, Certificate[] cerificateChain );`

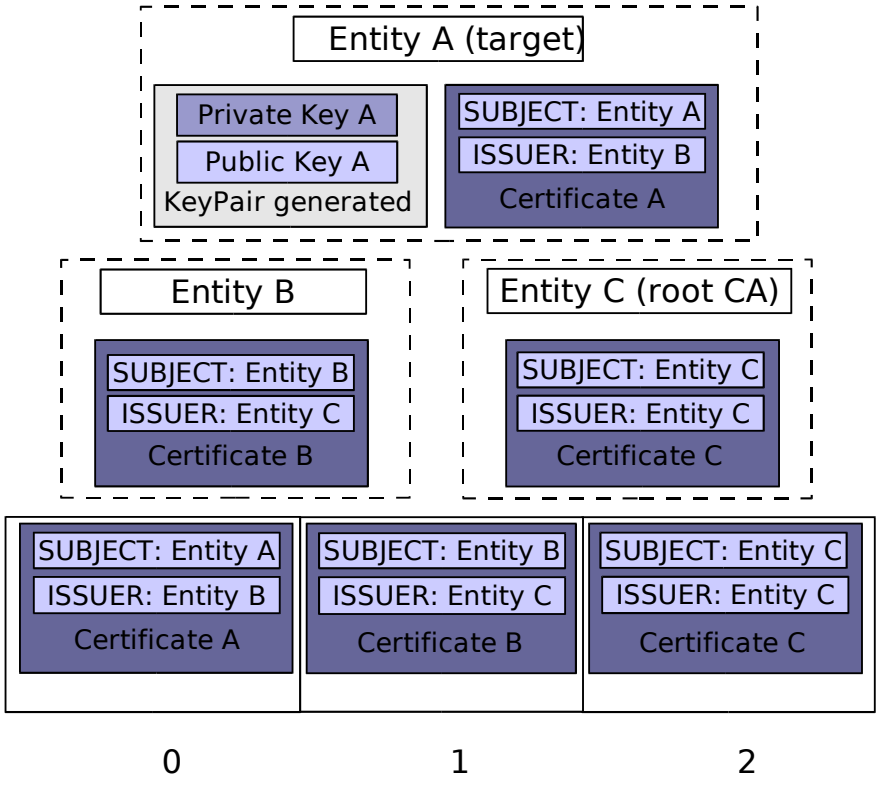
- `public SecurityWrapper ( PrivateKey key, Certificate[] cerificateChain, boolean verify );`

For the creation of this object the parameters passed in to the constructor can not be null objects (in case of the second parameter, the certificate array, it can not be empty). In case this rule is no accomplished, the constructor will throw a NullPointerException exception, and obviously, the object will not be created.

The parameters which must be passed in to the constructor are the following:

- PrivateKey: It is the private key that will be used for signing the serialized object into the SignedMarshalledObject. This key must be the same with which the jar file that contains the class files (needed for unmarshal the marshaled object) has been signed or will be signed.

- CertificateChain: It is the certificate chain that will be used for authentication and checking the proxy integrity when it is dynamically loaded from the target entity. The certificate chain is a sorted array of certificates, where the target certificate (it contains the public key associated to the private key passed as first parameter) is the first element in the array, and the root certificate (certificate that certifies all previous certificates in the chain) is the last element in the array.



Well-formed certificate chain (Array of certificates)

The third optional parameter which may be passed in to the constructor allows to check if the certificate chain passed in is a well-

formed certificathe chain. If the boolean value passed is `true`, the verification will be made, while if the boolean value is `false` this verification will not be made.

If the verification fails, the constructor throws an exception signaling that the certificate chain is not well-formed, and the creation of SecurityWrapper object will not be completed.

Helper classes for certificates and keys manipulation

To do easy the certificate and key retrieving from keystores, a helper class has been implemented:

- `net.fada.directory.security.FadaSecurityHelper`

This class contains some specific methods for getting keys and certificates stored in keystores which exist in the local machine.

To retrieve a private key the following static method may be used:

- ```
  public static PrivateKey getPrivateKeyFromFile(
        String fileName,
        String password,
        String alias );
  ```

This method returns the private key stored in the keystore specified by the parameter “fileName”, and which is associated to the key entry specified by the alias name “alias”. To retrieve this private key is necessary pass in to the method the password which protects the access by unauthorized personnel.

This method returns the private key if and only if the following conditions are held:

- The file specified by the parameter “fileName” exists
- The file specified by the parameter “fileName” has read permissions for the user executing the application.
  ```

- There is a key entry associated with the alias name passed in as parameter.
- The private key stored in the keystore has a key algorithm that is known by the key manager of the standard JAVA API.

If any of these conditions can not be accomplished the method will throw an exception signaling that the key is unrecoverable, detailing the reason.

In a similar manner, to get a certificate stored in a keystore, the following static method may be used:

- ```
  public static X509Certificate getCertificateFromFile(
        String fileName,
        String password,
        String alias );
  ```

The parameters passed in to the method are the same as the previous method. But the parameter “password” is not really required for the correct retrieving of the certificate. In this case a null object may be passed in for this parameter.

The conditions which must hold true to complete the retrieval successfully are nearly the same:

- The file specified by the parameter “fileName” exists
- The file specified by the parameter “fileName” has read permissions for the executing user.
- There is a certificate entry associated with the alias name passed in as parameter.
- The certificate stored in the keystore has a certificate algorithm that is known by the certificate manager of the standard JAVA API.

If any of these conditions can not be hold true the method will throw an exception signaling that the certificate is unrecoverable, detailing the reason.
  ```

Default FADA node security wrapper implementation

In the default implementation of a FADA node, the security wrapper functionality consists in the following parts:

- An optional secure communication which is provided through the use of HTTPS messages between the FadaProxy and Fada node.
- Secure web administration using HTTPS messages and SSL sessions.

Secure communications between Fada node and its proxy

A new implementation of Fada proxy has been written to allow secure communications between a Fada node and its proxy. To do this, a new class has been implemented:

- `net.fada.directory.FadaSecureProxy`

This class allows the communication with a Fada node in a secure way. All messages sent by this class to the Fada node go through HTTPS using the SSL protocol.

SSL is a technology to ensure privacy and reliability in the communication between two applications. It uses an asymmetric cryptographic system based in public/private key for negotiating a key used to establish a communication based in symmetric encryption. SSL is the encryption protocol most widely extended and used in Internet nowadays. Moreover, it is the protocol most used in web servers where confidencial information is requested.

The main security properties provided by SSL are:

- Secure communication based in symmetric encryption.
- Authentication and negotiation based in asymmetric encryption.

- Reliable communication based in message integrity.

Among the different communication options there is the possibility to authenticate the partakers of the connection through the use of certificates. These certificates can be verified by a verifier entity as VeriSign. Although, it is also allowed that the partakers are not authenticated. So, the following communication modes can take to:

- Anonymous communication: None of the partakers is authenticated.
- Server authenticated.
- Client authenticated.
- Both authenticated.

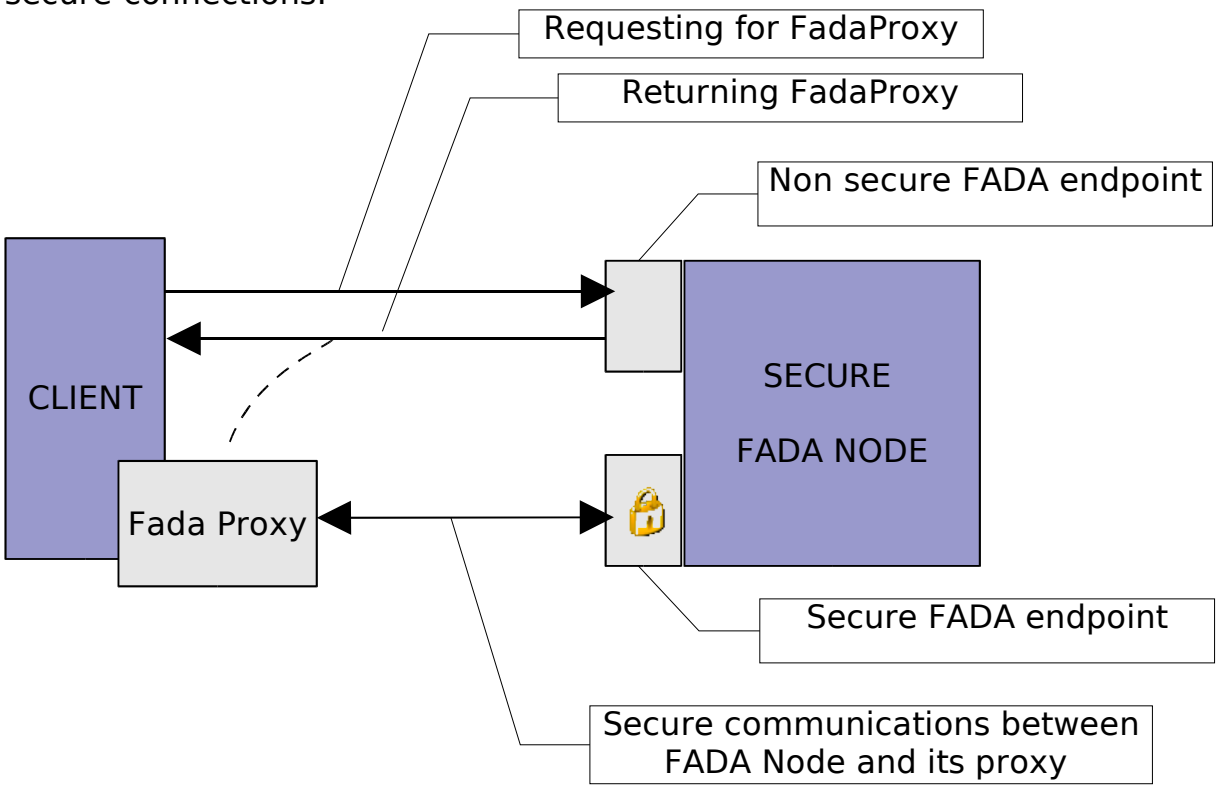
In the Fada secure communications, both are authenticated. To achieve this, both must have certificates based in private/public key (like X509 certificates). If some of the partakers doesn't accomplish with this rule an insecure communication will be assumed.

But a client who would like to communicate with a Fada node will first need to retrieve the Fada proxy (FadaSecureProxy) associated with this Fada node. To take on this bootstrapping action, no secure communication is required. So, the Fada node must allow two communication modes: secure and insecure mode. In the design of the secure Fada node the following functionality has been taken in account:

- To listen requests via insecure communication:
  - Only the following GET HTTP requests are allowed:
    - State queries (ping, time, view neighbors, view services, ...).
    - Non-administrative web queries (view documentation, view images, ...).
    - Proxy queries (to retrieve the Fada proxy).

- To listen requests via secure communication.
  - Both POST and GET HTTP requests are allowed:
    - State queries.
    - Administrative and non administrative web queries.

In a secure Fada node, two ports are ready to listen for queries. A port takes over insecure connections, while another port takes over secure connections.



Notes about policy files configuration

Java policy files

Java security has always been an issue, especially for networked code. While it has always been possible to develop custom security policies to protect private resources (using Java's security-manager paradigm), this model didn't easily allow for flexible policies. Up until now, enforcing such policies seemed somewhat impractical. With the advent of JDK 1.2, the new security model improves greatly (not by replacing the original model, but by enhancing it).

Java has always had many different faces to its security model. It has a strongly typed compiler to eliminate programming bugs and help enforce language semantics, a bytecode verifier that makes sure the rules of Java are followed in compiled code, a classloader that's responsible for finding, loading, and defining classes and running the verifier on them, and the security manager: the main interface between the system itself and Java code.

There are two default policy files concerning to security into JVM and which are inside any JDK distribution:

- `java.security` (located in *<java_install_dir>/jre/lib/security* directory)
- `java.policy` (located in *java_install_dir>/jre/lib/security* directory)

The first of them makes reference to various security properties which are set for use by java.security classes. This is where users can statically register Cryptography Package Providers ("providers" for short)[1]. This file is not necessary to be changed, unless a customized implementation of a FADA node is made, and it uses a cryptography algorithms that are not supported (implemented) by default providers.

1 The term "provider" refers to a package or set of packages that supply a concrete implementation of a subset of the cryptography aspects of the Java Security API. A provider may, for example, implement one or more digital signature algorithms or message digest algorithms.

The java.policy file is the default name given the *user* security policy. By default, this policy file is stored in the user's home directory. Since policy files only grant privileges, there is no danger of clashing. In other words, there is no provision for denying a privilege except to simply not grant it.

Fada policy file

The fada policy file is stored in *<fada_install_dir>* and is called `policy.file`. The following requirements or permissions must be allowed in the execution of a Fada node:

- **file read permissions** for the following directories (and files contained in them):
  - *<root_fada_dir>*
  - *<root_fada_dir>*/docs
  - *<root_fada_dir>*/dl
  - *<root_fada_dir>*/images
  - *<root_fada_dir>*/policy.file
  - *<tmp_dir>*
- **file write permissions** for the following files:
  - *<root_fada_dir>*/fada.rc
  - *<tmp_dir>*
- **file delete permissions** for the following files:
  - *<root_fada_dir>*/fada.rc
  - *<tmp_dir>*
- **socket permissions**
  - To allow listen, accept, resolve incoming and outgoing connections at non-secure port

- To allow listen, accept, resolve incoming and outgoing connections at secure port (only if the Fada node is configured to work in secure mode)
- To allow connect to neighbor proxies codebase resources

- **Runtime permissions**
  - To allow create class loaders
  - To allow get class loader
  - To allow exit to Virtual Machine (JVM)
  - To allow set Security Manager
  - To allow create threads and modify their states
- **Property read access permissions**
  - To allow read java.rmi.server.codebase property
  - To allow read `http.proxyHost` property
  - To allow read `http.proxyPort` property
  - To allow read `java.io.tmpdir` property
- **Property write access permissions**
  - To allow write `java.rmi.server.codebase` property
  - To allow write `http.proxyHost` property
  - To allow write `http.proxyPort` property
  - To allow write java.security.policy property

The policy file used to run a Fada node is fully wrote in appendix A

Velocity and FADA Web based management

In its version 5.2.4, FADA improved a lot the web based management module using as HTML rendering engine the Jakarta Velocity Java-based template engine.

The manner to create dynamicaly these pages was very improper before to adopt this technology for rendering the few HTML pages that conforms the FADA web based management. The pages were created using hardcoded string concatenation... Incredible but true! This way to construct the pages was very unflexible, uncustomizable and for each minimal change in these pages implied that the code must be recompiled.

Using Velocity, the web based management module achieves the following characteristics:

- Implements MVC model.
- Based on templates: Code recompilation after some change in a page is not needed never again.
- Customizable by node administrator (even in runtime).

It is important to acquire a minimal knowledge level about Velocity technology to understand properly the concepts explained in the next chapters.

Templates and variables

For the actual version of FADA there are 13 templates used to shape its web based management. Each template is asociated to a particular URI and a set of variables to be used by them. These variables are accessible by templates due they are in the velocity engine context.

There are a few global variables that can be used by all the templates, due are defined during velocity engine initialization (and not modified never again). These variables is shown in the next table:

Variable name	***Description***	***Value sample***
Node_Public_URL	String containing the host:port value where FADA node is binding	`slacky.fadanet.org:2002`
FadaVersion	String containing the version of the	`5.2.6`
LOGO_NAME	Filename of the picture choosed as logo for the FADA node	`fadalogo.gif`
Node_ID	FadaID for the actual instance of the node	`5b553575-4d8f-734f-1e62-8ba006f7dc7e`

There are defined a virtual template names for each template. This virtual template names are associated to a physical template files by a properties file. This properties file is placed by default in the path:

- <FADA_INSTALL_DIR>/templates/templates.properties

All physical templates file are located in the same directory than the properties file.

This properties file is pointed by a property defined inside the FADA configuration file (*fada.rc*). It allows to retrieved the velocity properties file by the process in charge of construct and initialize the Velocity engine. This key name of this property is: *templatesPropertiesFile*.

The contents included in the templates.properties file complies with the properties file format specified to serve as initialization[2] parameter to the Velocity engine.

In the next table, it is shown the relationship between URI and template associated. All cited templates in the table are available in any FADA bundle since its 5.2.4 version.

URI	***Virtual Template***	***Default template file name***
/	MAIN_PAGE	Main.vm
/admin	ADMIN_PAGE	AdminPage.vm

2 See `org.apache.velocity.app.Velocity.init(Properties props)` javadoc and other related cdocumentation

URI	***Virtual Template***	***Default template file name***
/services	SERVICES_PAGE	ServicesPage.vm
/neighbors	NEIGHBORS_PAGE	NeighborsPage.vm
/time	TIME_PAGE	TimePage.vm
/ping	PING_PAGE	PingPage.vm
/relatedDocs	DOCS_PAGE	DocsPage.vm
/admin/changePassPage	PASSWORD_PAGE	PasswordPage.vm
/plugins	PLUGINS_PAGE	PluginsPage.vm

The specific variables accesible by each template is shown in the next table:

Virtual Template	***Variable name***	***Variable description***
ADMIN_PAGE	FadaNeighbors	Array of *String* containing the neighbors registered in the node (*format host:port*)
	FadaNeighborsCount	Number of neighbors
	FadaServices	*FadaServiceMatches*[3] instance containing all the service proxies registered in the node
	AdminLastMessage	*String* with the last message generated as result to perform an administrative action
	PluginsCount	Number of plugins registered
	PluginsObjs	Array of *PluginIO*[4] objects containing the plugins registered

3 `net.fada.directory.tool.FadaServiceMatches`

4 `net.fada.plugins.PluginIO`

Virtual Template	***Variable name***	***Variable description***
SERVICES_PAGE	FadaServices	*FadaServiceMatches*[5] instance containing all the service proxies registered in the node
NEIGHBORS_PAGE	FadaNeighbors	Array of *String* containing the neighbors registered in the node (*format host:port*)
	FadaNeighborsCount	Number of neighbors
TIME_PAGE	Node_Time	String containing the value of actual time[6]
PING_PAGE	-none-	
DOCS_PAGE	-none-	
PASSWORD_PAGE	-none-	
MAIN_PAGE	-none-	
PLUGINS_PAGE	PluginsCount	Number of plugins registered
	PluginsObjs	Array of *PluginIO*[7] objects containing the plugins registered

Web based management: Themes

To create a theme for the FADA web based managemenent only is necessary to follow these steps:

- Recreate all the templates pages with the whished customized look.
- Place all the templates files in the same directory level.
- Creates a new templates.properties file in which is specified the new relationship between virtual template names and the recently created template pages

5 `net.fada.directory.tool.FadaServiceMatches`

6 Obtained value by calling to `System.currentTimeMillis()` method

7 `net.fada.plugins.PluginIO`

NOTE: In case of the new theme's templates are using CSS (set aside) or other resources for defining the page, these resources should be placed in the *<FADA_INSTALL_DIR>/dl* directory[8] .

To change the theme used with the web based management just modify the FADA configuration file to change the *templatesPropertiesFile* property value. So, the value of this property will contain the relative path to the templates.properties file created for the new theme.

8 This is the virtual context root for the mini HTTP server implemented inside FADA to serve the downloable resources as JAR file containing the classes of the FADA proxy (*fada-dl.jar*).

Fada Plugin Architecture

To permit add extensions to the functionality present in a FADA node by a modules written by third parties we are working to implement a plugin architecture module to be included to FADA.

Actual version 5.2.6 (and above) of FADA includes a beta implementation of this module corresponding to the first iteration's design.

The related documentation of this new component developed inside the scope of the FADA project is separated from this document because it is changing very often. For further information please refers to the last updated document "*Fada Plugin Architecture*".

Installation and Setup

The FADA software bundle comes in two flavors: a self-installing wizard for Windows platforms, and a self-installing shar script for UNIX platforms.

The installation of the software is straightforward, requiring only to execute the self-extracting archive on any directory.

There is also a configuration script that is automatically executed by the installation script or wizard, to allow the easy configuration of the FADA software, if it is intended to set up a permanent (or temporary) FADA node. If it is only desired to use the software to develop server or client classes, this step is irrelevant.

The configuration utility creates or modifies the file *fada.rc*, that contains the parameters needed by a FADA node to run. The parameters are specified in a single line in the file fada.rc, followed by an equals sign (=) and the parameter value, without any space. Such parameters are:

- The port number. This is the port where the FADA node will be listening for requests from outer entities. In theory, only two types of connections will be made to this port. One type is the connections performed by the Fada proxy, that invokes the methods on the FADA node to perform operations such as connection and disconnection of FADA nodes, and the registration and lookup of service proxies. The other type of connections will be those performed from within a web browser, to inspect the state of a FADA node, and to administer it. The parameter name is *nonSecurePort*.

- The url. This is the url that outer clients must use in order to access the instance of the FADA node installed. It must include the hostname, domain and port number of the FADA node. Note that this url needn't be the same url as the host it is running on, specially if a firewall exists in your environment. More on this issue a bit later. The parameter name is *nonSecureURL*.

- The type of FADA node. If an SSL port will be used for administration it must be specified by this parameter. The parameter name is *secureMode*, and it accepts the values *true* or *false*.

- The secure port. This parameter is optional, and is not provided in older versions of the FADA software, such as the ones that are publicly available at the time of this writing. This port can be used to administer the FADA node by using HTTP over SSL, that allows the use of encrypted connections. In this way neither the administrator password nor the operations performed can be easily seen by foreign entities. The parameter name is *securePort*. The parameter *secureMode* must be set to true to make this parameter work.

- The secure url. This parameter is also optional, and neither used in the older versions of the FADA software. As the port number for SSL connections is different, so must be the url. The secure port must also be specified. The parameter name is *secureURL*. The parameter *secureMode* must be set to true to make this parameter work.

- The policy file. This parameter specifies the Java policy file to be used by the security manager the FADA node is executing. The format, syntax and options of this file can be seen in the document entitled *FETISH communications and FADA security wrappers*. That document also points to the authoritative sources of information about Java, security and policy files. The parameter name is *policy*.

- The codebase. This parameter specifies the url where the classes for the FADA proxy and related classes can be found. The FADA node provides a mini HTTP server that will be used to deliver these classes. However, an external HTTP server may be used if it is desired. The url of such server must be specified by this parameter. If the FADA embedded HTTP server is used, the codebase parameter must be the public url of the FADA node plus the path "*/dl/fada-dl.jar*". The protocol prefix must be specified. If, for example, the url of the FADA node is *fada.fadanet.org:2002*, then the codebase annotation specified in the configuration script must be *http://fada.fadanet.org:2002/dl/fada-dl.jar*. More on downloadable code and the codebase property in later chapters. The parameter name is *codebase*.

- The list of nodes to connect this node to initially. By specifying an initial list of nodes to connect this one to, it is not necessary to enter the administration interface to manually connect them. The FADA node will do that upon startup. Zero or one of these parameters can be specified, one for each node it is desired to connect this FADA node to. The parameter name is *connectTo*. The list of FADA nodes to connect to is separated by any number of spaces (ASCII char 32). If the list spans more than one line no carriage return characters must be added.

- The username and password of the administrator. Both the username and password must be concatenated together by introducing a colon and converted to base64 coding. If, for example, the username is "`fada`" and the password is "`password`", the concatenation is "fada:password", and the base64 codification of this string is "`ZmFkYTpwYXNzd29yZA==`". Note that the base64 coding scheme is case sensitive. The FADA software provides a base64 encoder/decoder in the class net.fada.directory.tool.Base64. It can be invoked like this:

  ```
  java -classpath fada-lib.jar net.fada.directory.tool.Base64 {encode|decode}
  ```

  and it will encode or decode, as specified by the parameter, all characters sent to its standard input to its standard output. The fada-lib.jar file is part of the standard FADA distribution. The parameter name is *administrator*.

- If the FADA node is to be run behind a firewall, and an http proxy is to be used to communicate with the rest of the world, the http proxy hostname and port number can also be specified. Note that, in spite of the use of http proxies there must be at least one port opened in the firewall and redirected to the FADA node. The parameter names are *proxyHost* and *proxyPort*. No protocol prefix must be specified.

- Enabling the multicast extensions. Setting the property *allowsMulticastDiscovery* to true implies to enable the multicast extensions in the FADA node. It says, during the startup of the FADA node the announcement mechanism is performed, and the node will be discoverable by FADA clients using the discovery mechanism.

- The multicast group. By default is set to *public*, but it is posible to create separated federations in the same LAN sharing different values for this property between groups of FADA nodes.

- The web based managemenent theme. By default, this property value contains the relative path to the default theme for the FADA web based managemenent. In case to wish used other installed theme, the value of this property must be changed to contain the relative path to the customized theme.

- The plugins. Plugins installed in the FADA node are specified in the file which is specified in the property *pluginsPropertiesFile*. To enable or disable a plugin refers to this file.

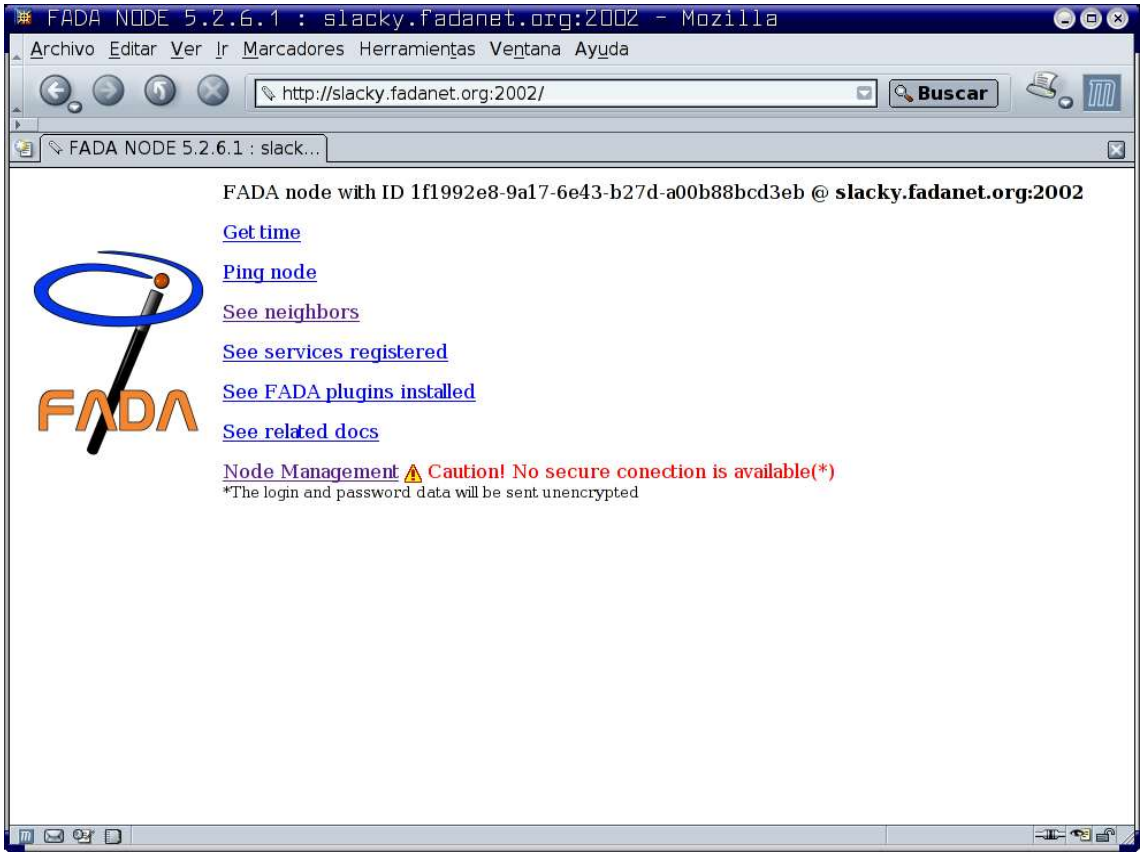
The fada.rc file may have comments. A comment is started by a hash mark (#), and spans through the rest of the line. Multiline comments are not allowed, multiple comment lines must be used if necessary. An example of the *fada.rc* file is given below:

```
# Example fada.rc file
# by the FADA development team
nonSecureURL=camaron.fada.net:2002
secureURL=camaron.fada.net:2003
nonSecurePort=2002
securePort=2003
secureMode=false
administrator=ZmFkYTpwcm9qZWN0
policy=./policy.file
codebase=http://camaron.fada.net:2002/dl/fada-dl.jar
allowsMulticastDiscovery=false
proxyHost=proxy.fada.net
proxyPort=80
connectTo=www.singladura.com:2002 fada.fadanet.org:2002
allowsMulticastDiscovery=true
multicastGroup=public
templatesPropertiesFile=./templates/templates.properties
pluginsPropertiesFile=./plugins/plugins.properties
# KeyStore=/home/jordi/certificates/b
# TrustStore=/home/jordi/FadaTrustStoreB
# aliasCA=thawte test ca root
```

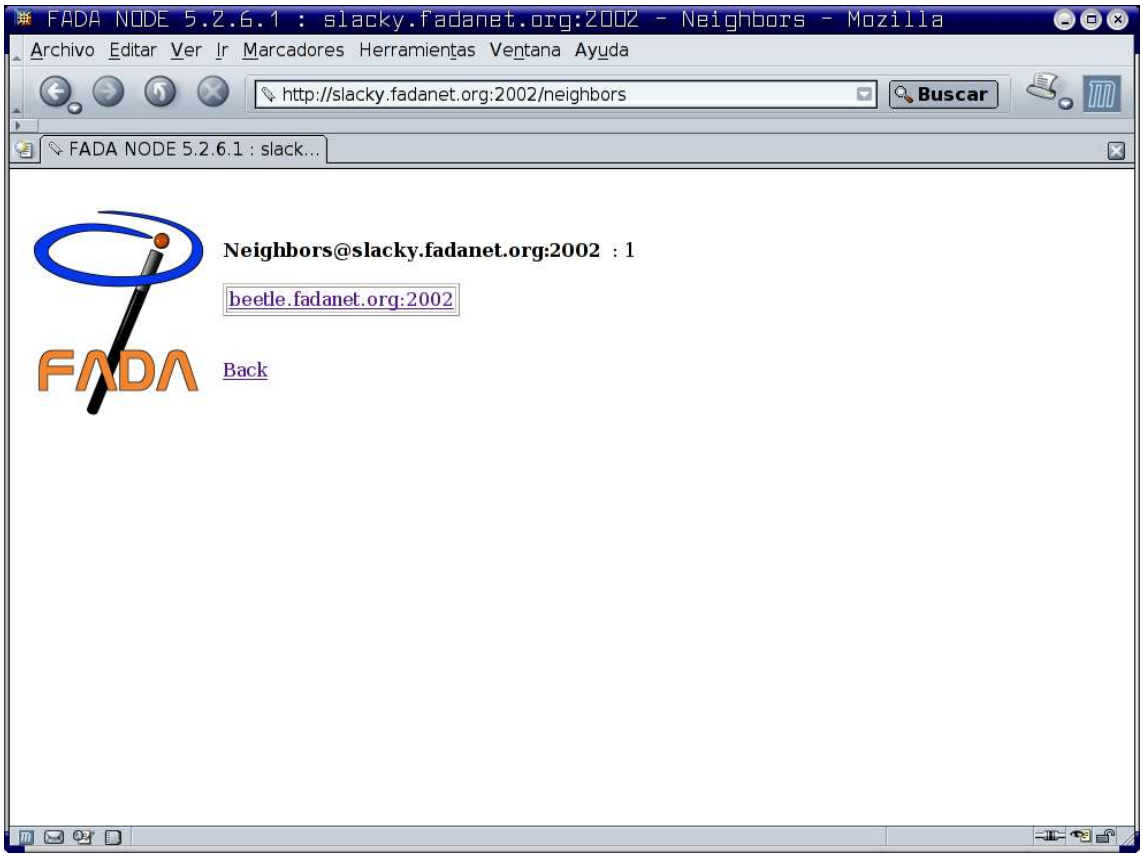
Administration of the FADA

Although a FADA node needs no administration, the management of the FADA network requires initial setup. This initial setup can be greatly automatized, and the FADA software provides the means for this automatization of the bootstrap procedure of the FADA node. However, if a finer control is desired on the FADA network, an administration front-end is provided.

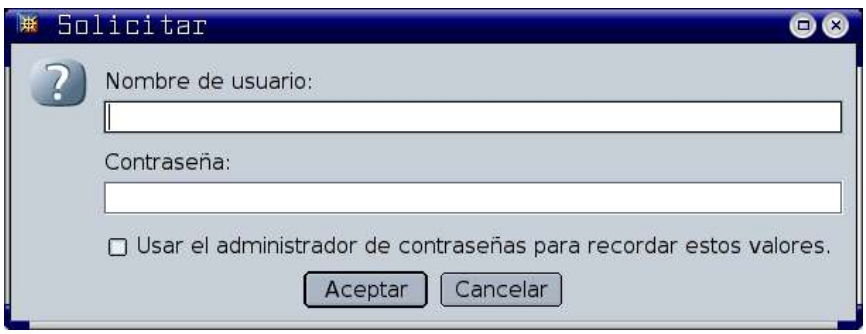
This administration front-end is accessible through a simple web browser. All it's needed is to provide the URL of the FADA node whose state is wished to monitor and/or manipulate. Starting a web browser and pointing it to, for example, slacky.fadanet.org:2002, we obtain the following screen:



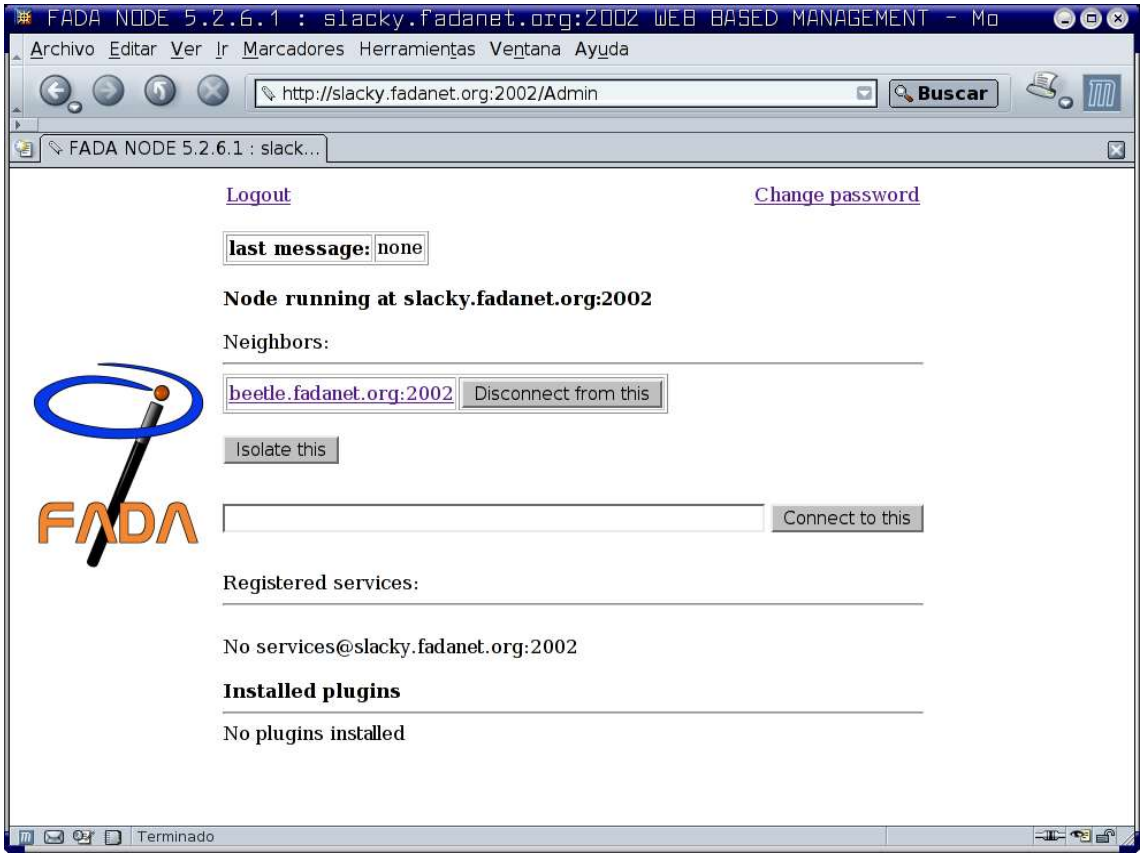
If the “See neighbors” link is clicked the current list of FADA nodes known to this node is shown, like this:



The administration profile can be entered by clicking on the “Node Management” link. This actions pops up an authentication window requiring a username and password to administer the FADA node:



Correctly entering the proper values the administration front-end is finally entered:



Through this interface it is possible to enhance or degrade the FADA node connectivity. In this way it is possible to create limited-range FADA networks.

NOTE: *The services registered in a FADA node (but available from all around the FADA architecture) can also be seen through the appropriate link, and they can also be deregistered from within this administration profile, as long as that the proper username and password are provided.*

Appendix A

A example of FADA policy file

```
grant codebase "file:ntp.jar"{
    permission java.net.SocketPermission "*:*",
"accept,listen,connect,resolve";};
grant codebase "file:fada-lib.jar"{
    // file permissions
    permission java.io.FilePermission "./docs/-", "read";
    permission java.io.FilePermission "./dl/-", "read";
    permission java.io.FilePermission "./images/-", "read";
    permission java.io.FilePermission "policy.file", "read";
    permission java.io.FilePermission "/tmp/*",
"read,write,delete";
    permission java.io.FilePermission "fada.rc",
"read,write,delete";
    permission java.io.FilePermission "tmp_fada_rc.tmp",
"read,write,delete";
    // socket permissions
    //permission java.net.SocketPermission "*:1024-", "connect";
    //permission java.net.SocketPermission "*:80", "connect";
    permission java.net.SocketPermission "127.0.0.1:1024-",
"accept";
    permission java.net.SocketPermission "172.26.0.3:1024-",
"accept,resolve";
    permission java.net.SocketPermission
"*:*","resolve,connect";
    permission java.net.SocketPermission "193.204.114.233:*",
"accept,listen,resolve,connect";
    permission java.net.SocketPermission "ntp2.ien.it",
"connect,accept,listen,resolve";
    // Runtime permissions
    permission java.lang.RuntimePermission "getClassLoader";
    permission java.lang.RuntimePermission
"getContextClassLoader";
    permission java.lang.RuntimePermission "createClassLoader";
    permission java.lang.RuntimePermission "exitVM";
```

```
    permission java.lang.RuntimePermission
"createSecurityManager";
    permission java.lang.RuntimePermission "modifyThread";
    permission java.lang.RuntimePermission "modifyGroupThread";
    // Property permissions
    permission java.util.PropertyPermission
"java.rmi.server.codebase", "read,write";
    permission java.util.PropertyPermission "http.proxyHost",
"read,write";
    permission java.util.PropertyPermission "http.proxyPort",
"read,write";
    permission java.util.PropertyPermission "java.io.tmpdir",
"read";
    permission java.util.PropertyPermission
"java.security.policy", "write";
};
```

Bibliography

Graph algorithms

[1] Leqiang Bai, Hajime Maeda: *A Broadcasting Algorithm with Time and Message Optimum on Arrangement Graphs*. Journal of Graph Algorithms and Applications. http://www.cs.brown.edu/publications/jgaa/ vol. 2, no. 2, pp. 1-17 (1998)

[2] P. Berthomé, A. Ferreira, S. Perennes: *Optimal Information Dissemination in Star and Pancake Networks.*

[3] M. Frans Kaashoek, Andrew S. Tanenbaum, Susan Flynn Hummel, Henri E. Bal: *An Efficient Reliable Broadcast Protocol*

[4] John Sucec, Ivan Marsic: *An Efficient Distributed Network-Wide Broadcast Algorithm for Mobile Ad Hoc Networks.*

[5] Yu-Chee Tseng, Wu-Lin Chang, Jang-Ping Sheu: E*fficient All-to-All Broadcast in Star Graph Interconnection Networks*. Proc. Natl. Sci. Counc. ROC(A) Vol. 22, No. 6, 1998. pp. 811-819.

Jini over Internet

[6]Ahmed Al-Theneyan, Piyush Mehrotra, Mohammad Zubair: *Enhancing Jini for Use Across Non-Multicastable Networks.*

The Kalman filter

[7] Peter D. Joseph: *Introductory Lesson to the Kalman Filter*.

[8] Peter S. Maybeck: *Stochastic models, estimation, and control. Volume 1.*

[9] Patrick D. O'Malley: *Use of a Kalman Filter to Improve Realtime Video Stream Image Processing: An Example.*

[10] Phillip D. Stroud: *A Recursive Exponential Filter For Time-Sensitive Data.*

[11] Greg Welch, Gary Bishop: *An Introduction to the Kalman Filter.*

[12] Greg Welch, Gary Bishop: *SCAAT: incremental Tracking with Incomplete Information.*

Network delays and algorithms

[13] Rene L. Cruz: *A Calculus for Network Delay.*

[14] D.L. Mills: Internet Delay Experiments (RFC889).

Jini

[15] W. Keith Edwards: Core Jini. The Sun MicroSystems Press. Prentice-Hall, 1999.

RMI

[16] Esmond Pitt and Kathleen McNiff: The Remote Method Invocation Guide. Addison-Wesley 2001.

Java

[17] Tim Lindholm, Frank Yellin: The Java Virtual Machine Specification, second edition. http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

Java based technologies

[18] Apache Jakarta Velocity guides:

Developers: http://jakarta.apache.org/velocity/developer-guide.html

Users: http://jakarta.apache.org/velocity/user-guide.html

Other

[19] IANA - Reserved multicast addresses.

http://www.iana.org/assignments/multicast-addresses

[20] RFC1945 – Hypertext Transfer Protocol – HTTP/1.0

[21] RFC2616 – Hypertext Transfer Protocol – HTTP/1.1

[23] RFC2617 – HTTP Authentication: Basic and Digest Access Authentication

[24] RFC1321 – The MD5 Message-Digest Algorithm

[25] RFC1341 – MIME (Multipurpose Internet Mail Extensions

[26] RFC1421 – Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures.

[27] RFC1510 – The Kerberos Network Authentication Service (V5)

[28] RFC1511 – Common Authentication Technology Overview

[29] RFC1704 – On Internet Authentication

[30] RFC1750 – Randomness Recommendations for Security

[31] RFC1760 – The S/KEY One-Time Password System

[32] RFC2002 – IP Mobility Support

[33] RFC2045 – Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.

[34] RFC2047 – MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII Text

[35] RFC2195 – IMAP/POP AUTHorize Extension for Simple Challenge/Response

[36] RFC2222 – Simple Authentication and Security Layer (SASL)

[37] RFC2243 – OTP Extended Responses

[38] RFC2289 – A One-Time Password System.

[39] RFC2444 – The One-Time-Password SASL Mechanism.

[40] RFC2518 – HTTP Extensions for Distributed Authoring – WEBDAV

[41] RFC 2459 - *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*.

[42] RFC2693 - *SPKI certificate theory*.

[43] RFC2246 - *The TLS Protocol Version 1.0.*

Note: all RFCs can be found at `http://www.ietf.org/rfc/rfcXXX.txt`*, where XXX must be substituted by the RFC number of interest. For example, the bibliography entry number 37 has the url* http://www.ietf.org/rfc/rfc2518.txt.

Security background

[44] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. In Jan Bosch, Jan Vitek, and Christian D. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, Lecture Notes in Computer Science volume 1603, pages 185 210. Springer, 1999.

[44] Matt Blaze, Joan Feigenbaum, and Jack Lacy. *Decentralized trust management*. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, pages 164 173, Oakland, California, May 1996.

[45] Sun Microsystems, Java 2 Security Architecture.

http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-specTOC.fm.html

[44] Sun Microsystems, Java 2 Security tools.

http://java.sun.com/j2se/1.2/docs/tooldocs/tools.html#security

[45] Sun Microsystems, Default Policy Implementation and Policy File Syntax

http://java.sun.com/j2se/1.4/docs/guide/security/PolicyFiles.html

[46] Bill Venners, Inside the Java Virtual Machine. Security. Ch. 3, McGraw-Hill, 1998.